Compact representations of pattern-avoiding permutations

László Kozma[1] and Michal Opler*[2]

[1]Faculty of Computer Science, TU Dresden, Germany
[2]Czech Technical University in Prague, Czech Republic

Abstract

Pattern-avoiding permutations are a central object of study in both combinatorics and theoretical computer science. In this paper we design a data structure that can store any size-n permutation τ that avoids an arbitrary (and unknown) fixed pattern π in the asymptotically optimal $\mathcal{O}(n \lg s_\pi)$ bits, where s_π is the *Stanley-Wilf limit* of π. Our data structure supports $\tau(i)$ and $\tau^{-1}(i)$ queries in $\mathcal{O}(1)$ time, sidestepping the lower bound of Golynski (SODA 2009) that holds for general permutations. Comparable results were previously known only in more restricted cases, e.g., when τ is *separable*, which means avoiding the patterns 2413 and 3142.

We also extend our data structure to support more complex geometric queries on pattern-avoiding permutations (or planar point sets) such as rectangle *range counting* in $\mathcal{O}(\lg \lg n)$ time. This result circumvents the lower bound of $\Omega(\lg n / \lg \lg n)$ by Pătraşcu (STOC 2007) that holds in the general case. For *bounded treewidth* permutation classes (which include the above-mentioned separable class), we further reduce the space overhead to a lower order additive term, making our data structure *succinct*. This extends and improves results of Chakraborty et al. (ISAAC 2024) that were obtained for separable permutations via different techniques. All our data structures can be constructed in linear time.

1 Introduction

A permutation $\tau = (\tau_1, \ldots, \tau_n)$ *contains* a permutation pattern $\pi = (\pi_1, \ldots, \pi_k)$ if there are indices $i_1 < \cdots < i_k$, such that $\tau_{i_j} < \tau_{i_\ell}$ if and only if $\pi_j < \pi_\ell$, for all $j, \ell \in [k]$, i.e., the subsequence $\tau_{i_1}, \ldots, \tau_{i_k}$ of τ is *order-isomorphic* to $\pi_1, \ldots, \pi_k$. Otherwise we say that τ *avoids* π.

Pattern-avoiding permutations arise in varied contexts and have been intensively studied for decades, both in combinatorics and in theoretical computer science (e.g., see [Kit11, Bón22]). For many concrete patterns π, the avoidance of π has natural alternative interpretations. For example, in one of the earliest results in this area, Knuth [Knu68, § 2.2.1] showed that 231-avoiding permutations are exactly those that are *sortable with a stack*. Monotone patterns (subsequences) are ubiquitous, e.g., in the context of the Erdős-Szekeres theorem. The question we ask in this paper is:

Can pattern-avoiding permutations be compactly represented?

One can obviously store a length-n permutation τ by storing τ_i for all $i \in [n]$. This can be done using $n\lceil \lg n \rceil$ bits, essentially matching the information-theoretic lower bound. However, to turn

*This work was co-funded by the European Union under the project Robotics and advanced industrial production (reg. no. CZ.02.01.01/00/22_008/0004590).

the representation into a useful data structure, one should support, preferably in constant time, both $\tau(i)$ and $\tau^{-1}(i)$ queries (given i, return τ_i, or given i, return j such that $\tau_j = i$), and perhaps other queries.[1] In other words, we would like to optimally compress a permutation in a way that allows accessing it *locally*, without fully decompressing it upon each query.

A simple solution is to store both $\tau(i)$ and $\tau^{-1}(i)$, for all $i \in [n]$, doubling the required space. An improved scheme that uses $(1+\varepsilon)n \lg n$ bits, for any $0 < \varepsilon < 1$, was given by Munro, Raman, Raman, and Rao [MRRR12], supporting $\tau(i)$ and $\tau^{-1}(i)$, and in fact, arbitrary power-k queries $\tau^k(i)$, in time $\mathcal{O}(1/\varepsilon)$. Golynski [Gol09] showed this tradeoff between space and query time to be essentially optimal.

Data structures whose space usage is optimal up to a constant factor are usually called *compact*; they have been studied extensively for storing permutations, trees, graphs, vectors, and other objects (e.g., see the textbook by Navarro [Nav16] for a broad survey).

Data structures with a stricter space requirement, having only a lower order additive overhead, are called *succinct*. Munro et al. [MRRR12] also design succinct schemes for representing permutations, i.e., using $n \lg n + o(n \lg n)$ bits, but the query costs in this case are necessarily super-constant.

All the mentioned solutions are, however, too wasteful for storing pattern-avoiding permutations. As the landmark result of Marcus and Tardos [MT04] shows, the number of length-n permutations that avoid a fixed pattern is *single-exponential* in n; this has long been known as the Stanley-Wilf conjecture.[2] This result is the main motivation for our work, as it allows, in principle, a data structure with *linear* space. More precisely, denoting by $\mathrm{Av}_n(\pi)$ the set of length-n permutations that avoid π, we have $|\mathrm{Av}_n(\pi)| \leq s_\pi^n$, for a quantity s_π known as the *Stanley-Wilf limit* of π. Note that s_π depends only on π, and can be defined as the limit $s_\pi = \lim_{n\to\infty} \sqrt[n]{|\mathrm{Av}_n(\pi)|}$ [Arr99].[3]

A natural goal is then to store a π-avoiding permutation τ compactly, i.e., in the asymptotically optimal space of $\mathcal{O}(\lg |\mathrm{Av}_n(\pi)|) = \mathcal{O}(n \lg s_\pi)$ bits, while supporting $\tau(i)$ and $\tau^{-1}(i)$ queries in time independent of n and π. In addition, the data structure that stores τ should be efficient to construct (ideally, in linear time). The reader may notice, that a data structure with these properties would allow *sorting* pattern-avoiding permutations in linear time. It follows from the Marcus-Tardos bound and a classical result of Fredman [Fre76] that a linear number of comparisons are sufficient for this task; however, an algorithm with actual linear running time (i.e., also including the overhead needed to find the comparisons) is far from obvious and was found only very recently [Opl24]. The data structure we aim for can thus be seen as a strengthening of this sorting result; to arrive at it, however, we will need a rather different set of techniques.

For some specific patterns π, permutations avoiding π have rigid structure. For instance, 231-avoiding permutations are preorder-sequences of binary search trees, and this underlying tree-structure can be used rather straightforwardly to compactly represent them.

Recently, Chakraborty, Jo, Kim, and Sadakane [CJKS24] gave a succinct data structure for the slightly more general class of *separable permutations*, supporting constant-time $\tau(i)$ and $\tau^{-1}(i)$ queries. Separable permutations are defined by the avoidance of the patterns 2413 and 3142 [BBL98]. These permutations also have a natural binary tree representation (they can be recursively constructed via the *sum* and *skew sum* operators [BBL98, CJKS24, BKO24]) and Chakraborty et al. exploit this structure to build their efficient representation.

Using related techniques, Chakraborty et al. also design a data structure for storing *Baxter-permutations*, although with polylogarithmic query times. Baxter permutations are not defined using

[1] We use the notation $\lg(x) = \log_2(x)$, $[k] = \{1, \ldots, k\}$, and we often write permutations inline, e.g., 231 for $(2, 3, 1)$. Both τ_i and $\tau(i)$ indicate the same value, but the latter notation is used when referring to a query.

[2] The proof of the conjecture by Marcus and Tardos [MT04] also builds on work by Füredi and Hajnal [FH92], and Klazar [Kla00].

[3] The growth rate of s_π as a function of $k = |\pi|$ is far from fully understood, it is known [Fox13] that $s_\pi \in 2^{O(k)}$ for all π, and $s_\pi \in 2^{\Omega(k^{1/4})}$ for some π.

classical pattern-avoidance, but via the related concept of *vincular patterns*. Baxter permutations are in bijection with floorplans/rectangulations [ABP06, FFNO11] and admit a representation using min-/max- Cartesian trees, which is explicitly used in [CJKS24].

These techniques are quite specific to separable and Baxter permutations. For general pattern-avoiding permutations a straightforward hierarchical structure is not apparent, and Chakraborty et al. leave open the question of efficiently representing other pattern-avoiding classes. Our main result addresses this question generally, for *arbitrary* avoided patterns.

Theorem 1.1. *Any permutation τ of size n that avoids a pattern π can be represented in $\mathcal{O}(n \lg s_\pi)$ bits, supporting $\tau(i)$ and $\tau^{-1}(i)$ queries in time $\mathcal{O}(1)$. Moreover, the representation can be constructed in $\mathcal{O}(n \lg s_\pi)$ time.*

To achieve this result, we use a *balanced decomposition* (partitioning or gridding) of permutations, related to merge sequences in the low *twin-width* regime. A similar technique was introduced in [BKO24], and we adapt and refine it here for the task at hand, also giving an efficient method for constructing it. The decomposition is fairly general, we believe that with its strong balance properties we introduce here, it may have further applications.

Our data structure also supports, in time $\mathcal{O}(1)$ and with negligible overhead, a host of other queries. These include (interval) *range minimum* (given i and j, return the smallest element in $\tau_i, \ldots, \tau_j$), *next smaller* (given i, return the smallest index $j > i$, such that $\tau_j < \tau_i$), and all their symmetries.

Like other related works, our data structure uses the Word RAM model with $\Theta(\lg n)$ word-length, space usage is however measured in bits, not words. We assume π to be fixed (i.e., not depending on n), but we emphasize that the query times do not hide a dependence on π. In the space and construction time bound we make the optimal $\lg s_\pi$ factor explicit, and note that for the space bound the $\mathcal{O}$-notation only hides a factor of two. Importantly, the avoided pattern π is not known during construction and operation, and only τ is given as input. We remark that the result circumvents Golynski's lower bound [Gol09] on the time-space tradeoff attainable for general permutations.

Reducing the constant factor in the space bound from two to one, and thus making the data structure succinct, is a challenging open question. We can achieve this for the case of *bounded treewidth* permutation classes, which include separable permutations as a special case.

More precisely, to each permutation τ, one can associate a graph G_τ, called its *incidence graph*. The vertices of G_τ are the pairs (i, τ_i) and vertex (i, j) is connected to the vertices $(i+1, \tau(i+1)), (i-1, \tau(i-1)), (\tau^{-1}(j+1), j+1), (\tau^{-1}(j-1), j-1)$ whenever these are well-defined. The *treewidth* of G_τ is then referred to as the treewidth of τ. This natural complexity-measure of permutations has played an important role in pattern matching algorithms (e.g., see [AR08, BKM21]).

Theorem 1.2. *Any permutation τ of size n and treewidth $t \in \mathcal{O}(1)$ that avoids a pattern π can be represented in $n \lg s_\pi + o(n)$ bits, supporting $\tau(i)$ and $\tau^{-1}(i)$ queries in time $\mathcal{O}(1)$. Moreover, the representation can be constructed in $\mathcal{O}_t(n)$ time.*

When τ is separable, its treewidth is known to be at most 7 [AR08], so Theorem 1.2 applies.[4] Note however, that since separable permutations are defined by *two* avoided patterns, the s_π constant for either of those would not capture the best possible constant factor in the space bound. However, we can extend Theorem 1.2 to any *supermultiplicative* class $\mathcal{C}$ of permutations of bounded treewidth, for a space bound of $\lg |\mathcal{C}_n| + o(n)$ where $\mathcal{C}_n$ denotes the set of length-n permutations in $\mathcal{C}$, thus obtaining a *succinct* structure also for separable permutations. Here, supermultiplicative means that for any i, j we have $|\mathcal{C}_{i+j}| \geq |\mathcal{C}_i| \cdot |\mathcal{C}_j|$. Note: separable permutations are counted by the Schröder numbers [Wes95], so their succinct data structure uses $n \cdot \lg(3 + 2\sqrt{2}) + o(n) \approx 2.54n$ bits. As in

[4] In fact, the more relevant complexity measure is *gridwidth*, and separable permutations are exactly those of gridwidth one. It is known that treewidth and gridwidth are within a small constant factor of each other [JOV18].

the case of Theorem 1.1, we can also support interval *range minimum*, *next smaller*, and similar queries in $\mathcal{O}(1)$ time. Previously, succinctly supporting these operations was only known for the separable special case [CJKS24], where some of the operations needed $\Theta(\lg \lg n)$ time.

We note that the query times in Theorem 1.2 depend on neither t nor $|\pi|$.

Adaptivity to pattern-avoidance. Our result can be seen as an algorithmic strengthening of the Stanley-Wilf, Marcus-Tardos [MT04] bound, turning it into an efficient data structure. This fits in a broader line of work studying the algorithmic consequences of pattern-avoidance, of which Knuth's stack sorting result can be seen as a first example. Fine-grained complexity bounds for pattern-avoiding inputs were obtained for several algorithmic problems. These include: searching in binary trees with rotation [CGK+15, CPY24, BKO24], the k-server and the traveling salesman problems [BKO24], factorization [BBGT24], permutation pattern matching and counting [BBL98, AR08, GM14, BKM21, GR22, EZL21], approximate counting [BEMS24], and sorting [Opl24].

In some of these results, pattern-avoidance is extended from permutations to planar point sets (under a general position assumption) in a straightforward way; this allows studying algorithmic problems on pattern-avoiding point sets. Our data structure for storing permutations can also be extended to support geometric queries such as the natural and well studied (rectangle) *range minimum*, or (rectangle) *range counting*.

In rectangle range minimum, we ask, given integers $a, b, c, d \in [n]$, for the smallest $i \in [c, d]$, such that $\tau^{-1}(i) \in [a, b]$. This is more general than (interval) range minimum where only left and right boundaries are given, and where sophisticated constant-time solutions exist [HT84, BV94, BFCP+05, FH11]. It is also natural to consider symmetric queries, i.e., we can ask, given integers $a, b, c, d \in [n]$, for the largest $i \in [c, d]$, such that $\tau^{-1}(i) \in [a, b]$, or for the smallest or largest index $i \in [a, b]$, such that $\tau(i) \in [c, d]$. In range counting, we ask, given $a, b, c, d \in [n]$, for the number of distinct indices $i \in [a, b]$, for which $\tau(i) \in [c, d]$.

Theorem 1.3. *Any permutation τ of size n that avoids a pattern π can be represented in $\mathcal{O}(n \lg s_\pi)$ bits, supporting rectangle range minimum and rectangle range counting queries in $\mathcal{O}(\lg \lg n)$ time. Moreover, the representation can be constructed in $\mathcal{O}(n \lg s_\pi)$ time.*

While it is convenient to state our results in terms of permutations, it is not difficult to extend them to arbitrary planar point sets, where both input points and queries are given by coordinates, e.g., as RAM words. The query times, again, are independent of the avoided pattern π.

In computational geometry, orthogonal range searching and its variants are among the best studied problems [TOG17, § 40]. For *rectangle range counting* in the plane, classical range-tree based data structures achieve $O(n \log n)$ space and $\mathcal{O}(\log n)$ query time. In the Word RAM model, assuming $\Theta(\lg n)$-bit words and polynomial-size coordinates, the query time has been improved to $\mathcal{O}(\lg n / \lg \lg n)$, with $\mathcal{O}(n)$ words of space [Cha88, JMS04]. Pătraşcu [Păt07] showed a matching lower bound on the query time in the *cell-probe* model, even assuming $\mathcal{O}(n \lg^{\mathcal{O}(1)} n)$ space. Our result circumvents this lower bound using the special properties of the input. Surprisingly, both *rectangle emptiness* and *rectangle range minimum* are solvable, in general point sets, with $\mathcal{O}(\lg \lg n)$ query times and $\mathcal{O}(n \lg \lg n)$, resp., $\mathcal{O}(n \lg^{\varepsilon} n)$ words of space [CLP11]; for these tasks our result only improves the space and preprocessing time bounds.

Sorting and compact data structures. Data structures for restricted classes of permutations have also been considered by Barbay and Navarro [BN13, Bar13]. The families of inputs they study come from classical *adaptive sorting*, e.g., permutations with few *contiguous sorted runs* or *interleaved sorted runs*, where the space- and time bounds depend on the distribution of run lengths. An input that consists of $k - 1$ interleaved sorted sequences clearly avoids $\pi = k, \ldots, 1$, our results thus subsume these cases. Moreover, permutations with few *contiguous* runs form a supermultiplicative class of bounded treewidth, our results thus imply a succinct data structure in this special case.

The connection between adaptive sorting and compact data structures observed in these works has also more broadly inspired our work. As mentioned already, efficient compact data structures yield efficient adaptive sorting algorithms; the reverse, however, is not known to hold generally, and has only been established in some special cases [Bar13]. Thus, the recent linear-time sorting algorithm for pattern-avoiding permutations [Opl24] suggests the natural question of whether a compact data structure exists for this family of inputs, affirmatively answered by Theorem 1.1.

Bounded twin-width permutations and matrices. Twin-width is a recently emerging complexity measure for permutations, graphs, as well as more general structures [GM14, BKTW21, BNdM+21]. Although we mostly avoid terminology related to twin-width to keep the presentation self-contained, our results and techniques closely relate to this concept.

In particular, it is known that a permutation τ that avoids π, as well as its matrix M_τ, have twin-width at most $2^{O(|\pi|)}$ [GM14, Fox13]. We could therefore directly use a recent compact data structure for storing low twin-width matrices, by Pilipczuk, Sokołowski, and Zych-Pawlewicz [PSZ22]. While it represents a more general class of objects, this data structure would fall short of our goals in several aspects: its query times are $\mathcal{O}(\lg \lg n)$, supporting only a subset of the queries we consider, taking superlinear time to construct. More importantly, the $\mathcal{O}$-notation in the space and query-time bounds involves an exponential dependence on twin-width; in our setting that would mean a doubly-exponential dependence on $|\pi|$. In contrast, in our data structure the query times are independent of π and the space bound involves an optimal $\lg s_\pi \in \mathcal{O}(|\pi|)$ factor. Nonetheless we adopt some terminology related to matrix-divisions from [PSZ22], and note some similarity in techniques, particularly in the proof of Theorem 1.3. In turn, our results also directly extend to storing permutations of *bounded twin-width* since every permutation of twin-width d avoids a certain pattern of length $(d+1)^2$ [GM14].

Corollary 1.4. *Any permutation τ of size n and twin-width $d \in \mathcal{O}(1)$ can be represented in $\mathcal{O}_d(n)$ bits, supporting $\tau(i)$ and $\tau^{-1}(i)$ queries in time $\mathcal{O}(1)$. Moreover, the representation can be constructed in $\mathcal{O}_d(n)$ time.*

A related very recent result is that π-avoiding permutations can be expressed as a composition of $\mathcal{O}_\pi(1)$ many separables [BBGT24]. Thus, by concatenating data structures for these separable permutations, one could represent a π-avoiding permutation, supporting some of the queries that we consider; in this way, however, the construction time, as well as space- and query time bounds would involve a *doubly-exponential* dependence on $|\pi|$. We especially stress that our data structure supports queries in $\mathcal{O}(1)$ time independent of the twin-width.

High-level description of our techniques. At a high level, our main data structure decomposes a permutation at two different levels of granularity. One can think of these levels as "merging" groups of neighboring rows or columns of the input permutation (viewed as a matrix), where the groups are of roughly equal, and carefully optimized size.[5] Due to the pattern-avoiding properties of the input, both levels of the decomposition have strong sparsity- as well as balancedness-properties (as defined in § 2, Lemma 2.2).

We briefly describe how a $\tau(i)$ query is answered (a $\tau^{-1}(i)$ query is analogous, as essentially all parts of our data structure are also stored in a duplicate, transposed form). We first locate the vertical strips of the decomposition where i falls, as well as the precise offsets within these; for this we use succinct tree and bitvector structures that allow navigating the levels of the decomposition.

To go from column- to row-coordinates, i.e., to compute τ_i from i, there are multiple challenges to overcome. As a key step, we store the sub-permutations induced by each vertical strip of the

[5] We note that our data structure significantly differs from previous schemes for storing general permutations, that are typically based on the cycle-structure of the input.

decomposition; this can be done compactly due to the fact that these permutations are themselves pattern-avoiding, and thus, the overall number of distinct permutations that arise can be tightly bounded. However, collapsing a vertical strip of the matrix into a permutation loses the precise row-information, so at first we can only get partial data from this structure: the index of the cell into which the entry (τ_i, i) falls in a sparse representation of the column, and the rank of the point within the permutation induced by its cell.

A separate data structure connects the cells at different levels of granularity and ultimately allows to identify the horizontal strips of the decomposition where (τ_i, i) falls. We convert the indices of these strips into their global vertical position by again using the succinct tree and bitvector structures. Finally, to obtain the precise vertical offset of the point *within* its horizontal strip, we consult a data structure that stores the permutations within horizontal strips, built analogously to those for vertical strips. This yields the value τ_i in $\mathcal{O}(1)$ time. We describe the details in § 3.

Extensions. To support the geometric queries (Theorem 1.3), we precompute more information at each level of the decomposition, from which rectangle-statistics can be reconstructed. For the original two-level decomposition, this would require too much space. We therefore construct $\Theta(\lg \lg n)$ levels instead, carefully controlling their granularity and the branching factor between consecutive levels. In this way, the amount of data stored at each level is kept small, while the running time is bounded as $\mathcal{O}(\lg \lg n)$. We describe the data structure in § 4.

For permutations of bounded treewidth (Theorem 1.2) we again use only two levels of the decomposition. However, we avoid separately storing both horizontal- and vertical strips. Instead we decompose the permutation into *components* – groups of cells in the decomposition that contain input points and that are connected by visibility. Components can reach multiple horizontal and vertical strips, and they contain *all* input points in those strips. Low treewidth guarantees that they are of bounded size; the two properties together allow storing sufficient detail for each component that allows converting between row- and column-information. We refer to § 5 for details.

Open questions. For our main result (Theorem 1.1), we leave open the question of a factor-two improvement in the space bound, i.e., whether the scheme can be made succinct. This factor is inherent in our design, due to the need to store both column-, and row-permutations, even for answering just $\tau(i)$-queries. We also leave open whether $\tau^k(i)$ queries can be supported within similar bounds.

For our geometric data structure (Theorem 1.3), can the cost of range counting queries be improved to $\mathcal{O}(1)$, or is there a non-trivial lower bound, or optimal trade-off result? In geometric applications, *weighted* versions of range searching/counting are often considered. Incorporating weights in our solution is problematic, since we store parts of the input only up to isomorphism. Informally, if we supported weights in full generality, then these could "encode" arbitrary permutations, and the classical lower bounds would apply. In a different direction, extending our results to a dynamic setting also appears challenging.

Can efficient compact representations be obtained for other interesting classes of permutations, not defined by pattern-avoidance? More broadly, can a compact data structure for a family $\mathcal{P}_n$ of permutations be obtained whenever sorting in $\mathcal{O}(\lg |\mathcal{P}_n|)$ time is possible, or is there a separation between the two problems for some natural $\mathcal{P}_n$?

2 Preliminaries

Permutation pattern-avoidance. Let $\tau = (\tau_1, \ldots, \tau_n)$ be a permutation. We call τ *non-trivial* if $n > 1$. We can bijectively map τ to an $n \times n$ (permutation) matrix M_τ where the τ_i-th entry of the i-th column is 1 for all $i \in [n]$ and all other entries are 0. We refer to matrix entries by row first

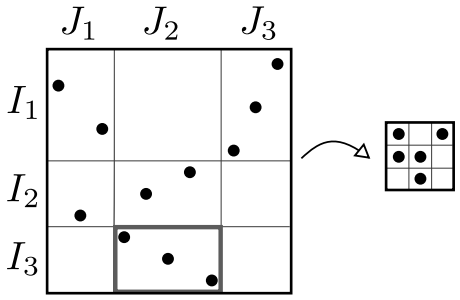

Figure 1: Permutation matrix M of size 11×11 (left) with division $(\mathcal{R}, \mathcal{C})$ and $M(\mathcal{R}, \mathcal{C})$ (right), with $\mathcal{R} = (I_1, I_2, I_3)$, $\mathcal{C} = (J_1, J_2, J_3)$, and cell $M[I_3, J_2]$ framed. Here, $I_1 = [1, 5], J_1 = [1, 3], I_2 = [6, 8], J_2 = [4, 8], I_3 = J_3 = [9, 11]$. Dots correspond to 1-entries, empty spaces to 0-entries.

and column second, ordering top to bottom and left to right. We also frequently refer to a 1-entry (τ_i, i) as a *point*. The statements "matrix M_τ contains/avoids matrix M_π" are interpreted naturally to mean "τ contains/avoids π".

A general 0-1 matrix M *contains* M_π or simply M *contains* π, if M_π can be obtained from M by deleting rows and columns and changing 1-entries to 0. Otherwise we say that M *avoids* M_π or simply M *avoids* π.

For a permutation π, we denote by $\mathrm{ex}_\pi(n)$, the maximum number of ones in an $n \times n$ 0-1 matrix that avoids π. Marcus and Tardos [MT04] showed that for every fixed π, the quantity $\mathrm{ex}_\pi(n)$ is linear in n. More precisely, there exists a quantity c_π so that:

Lemma 2.1. *Every $n \times n$ 0-1 matrix M with at least $c_\pi \cdot n$ one-entries contains π.*

The optimal value c_π is known as the *Füredi-Hajnal limit* of π, after the earlier conjecture [FH92], and can be defined as $c_\pi = \lim_{n\to\infty} \mathrm{ex}_\pi(n)/n$. It is known that the Füredi-Hajnal limit and the Stanley-Wilf limit (defined in § 1) are polynomially related; concretely, Cibulka [Cib09] showed $s_\pi = \Omega(c_\pi^{2/9}) \cap \mathcal{O}(c_\pi^2)$.

Divisions and coarsening. Let M be an $n \times n$ 0-1 matrix. Let $\mathcal{R} = (I_1, \ldots, I_k)$ be a partition of $[n]$, i.e., $I_1 \cup \cdots \cup I_k = [n]$ and $I_1, \ldots, I_k$ are pairwise disjoint *contiguous intervals*. Similarly, let $\mathcal{C} = (J_1, \ldots, J_{k'})$ be a partition of $[n]$. Then, we refer to $(\mathcal{R}, \mathcal{C})$ as a *division* of M; intuitively, the intervals in $\mathcal{R}, \mathcal{C}$ capture neighboring sets of rows and columns in M that are *merged* together. The resulting matrix $M' = M(\mathcal{R}, \mathcal{C})$ is defined as follows. Assuming $\mathcal{R}, \mathcal{C}$ as above, M' has k rows and k' columns, and entry $M'(i, j)$ is 1 if at least one entry $M(x, y)$ with $x \in I_i$ and $y \in J_j$ equals 1, and $M'(i, j)$ is 0 otherwise. When clear from the context, by the i-th row, resp., j-th column of the division $(\mathcal{R}, \mathcal{C})$ we refer to the submatrix of the original matrix M defined by the rows I_i, resp., by the columns J_j. Accordingly, the *height* of the i-th row is $|I_i|$, i.e., the number of rows of the original matrix that are merged into row i of M'. Similarly, the *width* of the j-th column is $|J_j|$. By $M[I_i, J_j]$ we refer to the submatrix of M at the intersection of rows indexed by I_i and columns indexed by J_j. We also call this submatrix the *cell* (i, j) of the division $(\mathcal{R}, \mathcal{C})$. A cell is non-zero, if it has at least one non-zero entry. We illustrate some of these concepts in Figure 1.

We say that a division $(\mathcal{R}', \mathcal{C}')$ is a *coarsening* of $(\mathcal{R}, \mathcal{C})$ if $\mathcal{R}'$ can be obtained from $\mathcal{R}$ by successively replacing neighboring intervals by their union, and if $\mathcal{C}'$ can be similarly obtained from $\mathcal{C}$. In this case, we also say that $(\mathcal{R}, \mathcal{C})$ is a *refinement* of $(\mathcal{R}', \mathcal{C}')$. Notice that divisions form a partial order by the coarsening relation, where the division with a single $[n]$ interval for both rows and columns is the coarsest, and the division with singleton sets for both rows and columns (i.e., capturing the original matrix) is the finest (i.e., least coarse).

Data structuring tools. We use as building blocks the following compact data structures. Note that all these structures are static: once built, they are not modified. While we omit discussing the details of their construction, this takes linear time for all components and is subsumed by the $\mathcal{O}(n \lg s_\pi)$ time necessary for sorting the input τ.

- *sparse bitvector*: stores values $B[1], \ldots, B[n] \in \{0, 1\}$, supporting in $\mathcal{O}(1)$ time the operations:
  - $\mathsf{read}(B, i)$: returns $B[i]$,
  - $\mathsf{rank}(B, i)$: number of 1-bits in $B[1, \ldots, i]$,
  - $\mathsf{select}(B, i)$: smallest j such that the number of 1-bits in $B[1, \ldots, j]$ is i, i.e., the index of the i-th 1.

  The data structure uses $n\mathcal{H} + o(n)$ bits, where $\mathcal{H}$ is the *empirical entropy* of the bitvector, i.e., $\mathcal{H} = p \lg \frac{1}{p} + (1-p) \lg \frac{1}{1-p}$, where p is the fraction of 1-bits in $B[1, \ldots, n]$; notice that $0 \leq \mathcal{H} \leq 1$. For possible implementations with these parameters, we point to [Nav16, § 4] and references therein.

- *tree*: rooted, ordered tree with n nodes, supporting the following operations in $\mathcal{O}(1)$ time:
  - $\mathsf{parent}(v)$: parent of node v,
  - $\mathsf{child}(v, i)$: i-th child of node v,
  - $\mathsf{childRank}(v)$: number of siblings to the left of node v,
  - $\mathsf{leafSelect}(i)$: i-th leaf from the left,
  - $\mathsf{leafRank}(v)$: number of leaves to the left of node v, excluding the leaves in subtree of v.

  Standard implementations using $2n + o(n)$ bits can be found in [Nav16, § 8] and references therein.

- *function*: a mapping $A \to B$, with $A = [|A|]$ and $B = [|B|]$, where $|B|$ is assumed to fit in a machine word. The space requirement of a standard implementation is $|A| \cdot \lceil \lg |B| \rceil$ bits. (A stronger bound of $\lceil |A| \cdot \lg |B| \rceil + \mathcal{O}(1)$ bits is achievable, but not needed in our work.)

- *range minimum* data structure: processes an array $A[1, \ldots, n]$, so that, upon queries $\mathsf{rm}(a, b)$ for $a, b \in [n]$, it returns an index $i \in [a, b]$ for which $A[i] = \min A[a, \ldots, b]$. (For simplicity we assume array entries to be distinct.) Standard solutions are based on an equivalence between range-minimum queries over A and *lowest common ancestor* queries in the Cartesian tree built from A, e.g., see [HT84, BV94, BFCP+05, FH11]. It is possible to answer $\mathsf{rm}(a, b)$ queries in $\mathcal{O}(1)$ time, with a data structure using $2n + o(n)$ bits of space; crucially, the data structure can be implemented so that the array A does not need to be read during queries.

Main structural lemma. An important ingredient of our data structure is the following decomposition of a permutation (matrix). As mentioned, this decomposition is related to merge sequences of low twin-width permutations (e.g., [GM14]), significantly adapted for our purposes.

Lemma 2.2 (Balanced Decomposition)**.** *Let π be a non-trivial permutation with Füredi-Hajnal limit c_π and let M be a π-avoiding $n \times n$ permutation matrix.*

For integer parameters $1 < m_1 < m_2 < \cdots < m_\ell < n$, there exist divisions $(\mathcal{R}_1, \mathcal{C}_1), \ldots, (\mathcal{R}_\ell, \mathcal{C}_\ell)$ of M such that:

(a) The division $(\mathcal{R}_i, \mathcal{C}_i)$ is a coarsening of $(\mathcal{R}_{i+1}, \mathcal{C}_{i+1})$ for each $i \in [\ell - 1]$.

(b) $M(\mathcal{R}_i, \mathcal{C}_i)$ has exactly m_i rows and columns.

(c) *Each row in* $\mathcal{R}_i$ *and each column in* $\mathcal{C}_i$ *has height, resp., width at most* $40 \cdot n/m_i$.

(d) *Each row and each column in* $M(\mathcal{R}_i, \mathcal{C}_i)$ *has at most* $10c_\pi$ *non-zero cells.*

(e) *Each row in* $\mathcal{R}_i$ *and each column in* $\mathcal{C}_i$ *is obtained from merging at most* $40 \cdot m_{i+1}/m_i$ *rows of* $\mathcal{R}_{i+1}$*, resp., columns of* $\mathcal{C}_{i+1}$*, for* $i \in [\ell - 1]$.

Moreover, the divisions $(\mathcal{R}_1, \mathcal{C}_1), \ldots, (\mathcal{R}_\ell, \mathcal{C}_\ell)$ *can be computed in time* $\mathcal{O}((\lg c_\pi + 1) \cdot n + c_\pi \cdot \sum_{i=1}^{\ell} m_i)$ *even if* π *and its Füredi-Hajnal limit* c_π *are a priori unknown.*

We give a self-contained proof of Lemma 2.2 in § 6. A weaker statement of a similar flavor was obtained in [BKO24]. In particular, Lemma 2.2 differs in enforcing an exact number of columns and rows in part (b), an additional balancedness condition between the layers in part (e) and very efficient implementation.

3 Compact data structure for pattern-avoiding permutations

In this section, we describe the main data structure referred to in Theorem 1.1. Let τ be an input permutation of length n, and let $M = M_\tau$ be its corresponding permutation matrix. To simplify presentation, we omit floor and ceiling operators, and we assume that n is larger than some unspecified constant.

The data structure is based on two non-trivial levels of the balanced decomposition (Lemma 2.2). More precisely, we set $\ell = 2$ and find a decomposition with the parameters $1 < m_1 < m_2 < n$ where $m_1 = n/\lg^2 n$ and $m_2 = n/\sqrt{\lg n}$. We will sometimes refer to the resulting divisions $(\mathcal{R}_1, \mathcal{C}_1)$ and $(\mathcal{R}_2, \mathcal{C}_2)$ as the *coarse-*, resp., *fine* division, and we refer to their cells as 1-cells, resp., 2-cells.

We now list the components and their space requirements. See Figures 2 and 3 for illustration.

A. Column and row trees T_C, T_R**.** The *column tree* T_C and the *row tree* T_R capture the merges of the rows, resp. columns that yield the divisions of the balanced decomposition.

More precisely, the root of T_C corresponds to $[n]$, i.e., the set of all columns, the m_1 children of the root (level-1) correspond to the columns in the coarse division $(\mathcal{R}_1, \mathcal{C}_1)$, the m_2 nodes on level-2 of the tree correspond to the columns in the fine division $(\mathcal{R}_2, \mathcal{C}_2)$, where if node x is a parent of node y, then the column of x contains the column of y, and the ordering of siblings is the same as the ordering of columns in the matrix. Note that the n columns of M are not explicitly stored in the tree. The construction of the row tree T_R is entirely symmetric.

Both T_C and T_R are represented using the succinct tree implementation mentioned in § 2, allowing rich navigation queries. The number of nodes in both trees is $1 + m_1 + m_2$, the space requirement is thus $\mathcal{O}(n/\sqrt{\lg n}) \subseteq o(n)$.

B. Column and row indices I_C, I_R**.** The *column index* I_C is a bitvector over the n columns of the matrix M, with 1-entries for the start of each column in the fine division, i.e., $I_C[i] = 1$ if a column of $\mathcal{C}_2$ begins at the i-th column of M, and $I_C[i] = 0$ otherwise. Given an index $i \in [n]$ of a column in M, we can find the index of the $\mathcal{C}_2$ column containing it using a rank query over I_C. Conversely, given a column index in $\mathcal{C}_2$, we can find its start index in M with a select query over I_C. Analogously, we store a bitvector I_R over rows of the original matrix M, marking the start of each row in $\mathcal{R}_2$.

Both bitvectors have size n with $m_2 = n/\sqrt{\lg n}$ non-zero entries, their space requirement using the implementation mentioned in § 2 is thus $o(n)$.

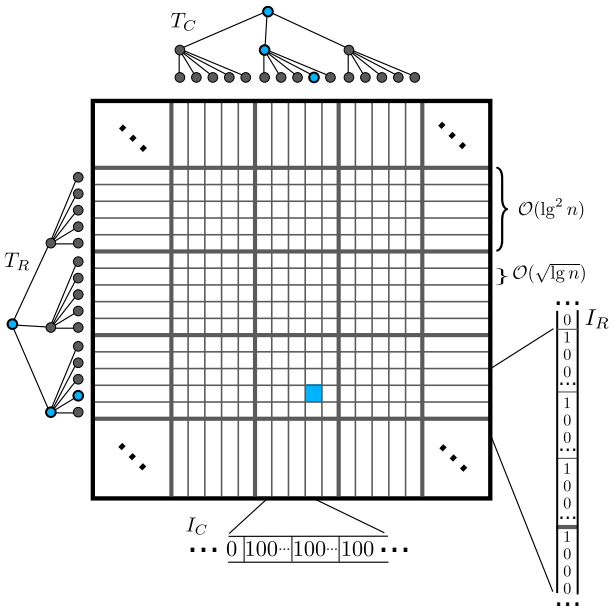

Figure 2: Decomposition of an $n \times n$ permutation matrix M with a (coarse) division $(\mathcal{R}_1, \mathcal{C}_1)$ that is a coarsening of a (fine) division $(\mathcal{R}_2, \mathcal{C}_2)$. Column and row trees T_C, T_R and column and row indices I_C, I_R illustrated. A 2-cell and corresponding nodes in T_R and T_C highlighted.

C. Column permutations colPerm**.** We store, for each $\mathcal{C}_2$-column C of M the *permutation* of the $|C|$ points of M that fall into C. Although not sufficient to get the exact row position for the entry in column i (which would immediately solve the $\tau(i)$ query), it allows to identify the $\mathcal{R}_2$-row index of the queried point, and relative rank information within the 2-cell where the query falls.

More precisely, the method colPerm invoked on a $\mathcal{C}_2$-column maps a column offset k to an index c (the non-zero cell index where the query falls), and a value v, the number of 1-entries that fall into the same cell c, below the queried point (τ_i, i). See Figure 3.

The cell index c refers to the c-th non-zero cell in the column, its domain is thus $[10c_\pi]$, by Lemma 2.2(d). We also store the inverse mapping denoted $\mathsf{colPerm}^{-1}$ that maps (c, v) to k.

Recall that there are $m_2 = n/\sqrt{\lg n}$ columns in $\mathcal{C}_2$ and the column width is at most $w = 40\sqrt{\lg n}$ by Lemma 2.2(c). Storing colPerm explicitly for each column would be too costly, we therefore implement a scheme that (i) stores the *permutation* of 1-entries that fall into the column, together with the partitioning of entries into cells, and (ii) uses global precomputed tables for each permutation up to isomorphism, for space efficiency.

The number of possible permutations of points induced by 1-entries in a column is $\sum_{k=1}^{w} s_\pi^k \in \mathcal{O}(s_\pi^w)$. (We use the fact that the permutation induced by the column is also π-avoiding.) The permutation is augmented with data about how many of the (up to) w entries fall into each of the (up to) $10c_\pi$ non-zero cells of the column. We refer to the permutation together with its augmentation as a *gridded permutation*. Notice that the number of possible distinct gridded permutations is $\mathcal{O}(s_\pi^w \cdot w^{10c_\pi})$.

Let $w_C = |C|$ denote the width of column C ($w_C \le w$). To implement colPerm for C, we store the *index* to the gridded permutation of length w_C in the column, this requires $\lceil \lg s_\pi^{w_C} \rceil + \lceil \lg w^{10c_\pi} \rceil \le w_C \lg s_\pi + 10c_\pi \lg w + 2 \in w_C \lg s_\pi + \mathcal{O}_\pi(\lg \lg n)$ bits (the first term is for the permutation, and the second for the gridding augmentation). In total, over all columns, this adds up to $n \lg s_\pi + o(n)$ bits, which will be accounted for in part D.

While for ease of notation we will simply write, e.g., $(c, v) \leftarrow \mathsf{colPerm}(k)$ or $k \leftarrow \mathsf{colPerm}^{-1}(c, v)$, this should be interpreted as obtaining a pointer to a permutation data structure stored in a precomputed table where each distinct gridded permutation is stored at most once.

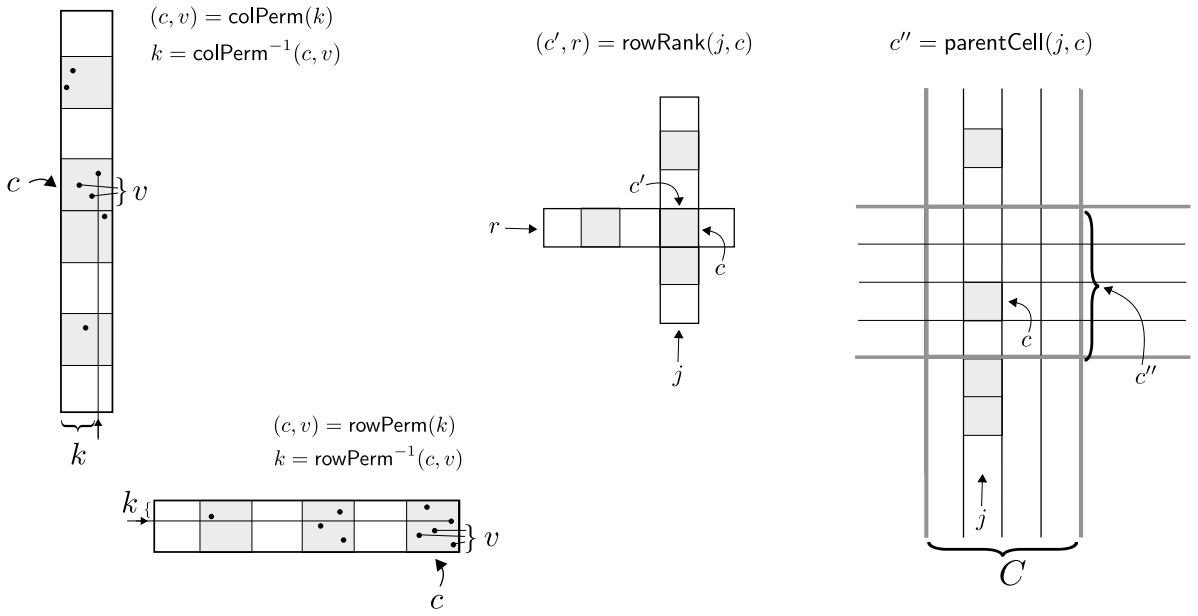

Figure 3: *Left*: column and row permutation mapping and their inverse. Value k is the absolute offset of the query within the column or row; c refers to the c-th non-zero cell within column or row (non-zero cells are shaded gray); v is the vertical rank of the query within its cell, i.e., v entries in cell are below query point.
Right: rowRank method: the c-th non-zero cell of the j-th column is the c'-th non-zero cell of the r-th row; parentCell method invoked for a $\mathcal{C}_1$-column C: the c-th non-zero 2-cell in the j-th $\mathcal{C}_2$-column in C is in the c''-th non-zero cell of C.

Although the number of possible gridded permutations to store is $\mathcal{O}(s_\pi^w \cdot w^{10c_\pi})$, we still cannot easily compute them upfront, not knowing π. However, we can examine each column C of $\mathcal{C}_2$ and only store the gridded permutations that actually appear in a column. First we can bucket permutations by size (i.e., the column width w_C), then by permutation order class (we can identify this by using the efficient pattern-avoiding-sorting [Opl24]), and finally by gridding. Storing gridded permutations of the same size in separate tables ensures that the index lengths correspond to the column widths and the above space bound holds. For all gridded permutations that appear, we build a simple data structure that explicitly stores the answer to every possible query on them, allowing $\mathcal{O}(1)$-time operations.

Recall that we need to map column offset k to cell index c and vertical rank v of the entry within its cell. This mapping has domain $[w]$ and range $[10c_\pi] \times [w]$. Storing it explicitly thus requires $\mathcal{O}(w \lg(c_\pi w))$ bits, and $\mathcal{O}(c_\pi w \lg w)$ for its inverse mapping. As we have at most $\mathcal{O}(s_\pi^w w^{10c_\pi})$ such structures, their total space requirement is $\mathcal{O}(s_\pi^w w^{10c_\pi} \cdot (w \lg(c_\pi w) + c_\pi w \lg w)) \subseteq o(n)$.

Additionally, observe that for any permutation class $\mathcal{C}$ that is a subclass of π-avoiding permutations, the scheme described encodes a column of width w' in $\lg |\mathcal{C}_{w'}| + \mathcal{O}_\pi(\lg \lg n)$ bits where $\mathcal{C}_{w'}$ is the set of all permutations in $\mathcal{C}$ of length exactly w'. This is a corollary of our approach that is oblivious to and independent of π except for the size of gridding given by the balanced decomposition. Over all columns, this adds up to at most $\lg |\mathcal{C}_n| + o(n)$ bits, assuming that $\mathcal{C}$ is supermultiplicative.

D. Recursive column structure G_C. This component is the crucial part of the data structure that allows converting from column to row data. It is a tree structure over the columns similar to T_C, but storing more detailed column-information from the input permutation matrix M.

Similarly to T_C, the root of G_C corresponds to $[n]$, i.e., the set of all columns, and the m_1 children of the root (level-1) correspond to the columns in the coarse division $(\mathcal{R}_1, \mathcal{C}_1)$, and the m_2 nodes on

level-2 correspond to columns in the fine division $(\mathcal{R}_2, \mathcal{C}_2)$. As we need to store other information at the nodes, we do not provide the flexible navigation methods that T_C offers. Instead, we only navigate the tree top-down, with the query $\mathsf{subColumn}(j)$ that gives a pointer to the j-th subcolumn of the current node. More precisely, for the root, $\mathsf{subColumn}(j)$ points to the j-th column in $\mathcal{C}_1$, and for a level-1 node corresponding to a $\mathcal{C}_1$-column, $\mathsf{subColumn}(j)$ points to the column in $\mathcal{C}_2$ that is the j-th within the current $\mathcal{C}_1$-column. Level-2 nodes (leaves) do not allow $\mathsf{subColumn}$ queries.

The key components of the interface are the methods $\mathsf{rowRank}(j, c)$ and $\mathsf{parentCell}(j, c)$, supported at both the root and level-1 nodes of the tree structure, and the method $\mathsf{colPerm}$, supported at the leaf (level-2) nodes; see Figure 3.

The first of these, $\mathsf{rowRank}(j, c)$ refers to the c-th non-zero cell of the j-th subcolumn of the column associated to the current node. Recall that both $\mathcal{C}_1$- and $\mathcal{C}_2$-columns consist of cells, of which at most $10c_\pi$ are non-zero, and we identify the c-th such non-zero cell. The call $\mathsf{rowRank}(j, c)$, returns a pair (c', r) that give row-information about the identified cell. More precisely, when called at the root, we find out that the identified 1-cell is the c'-th non-zero cell in the r-th $\mathcal{R}_1$-row. When called at a level-1 node, we find out that the identified 2-cell is the c'-th non-zero cell in its $\mathcal{R}_2$-row, and that this row is the r-th among the $\mathcal{R}_2$-rows within its parent $\mathcal{R}_1$-row.

The second method, $\mathsf{parentCell}(j, c)$ similarly identifies the c-th non-zero cell of the j-th subcolumn of the column associated to the current node, and returns c'', indicating that the coarser cell containing the current cell is the c''-th non-zero cell within its column. Notice, that this method is only useful when called at a level-1 node, where we learn about the 1-cell containing the current 2-cell. If called at the root, the coarser cell containing the 1-cell is necessarily the entire matrix.

Finally, for level-2 nodes (i.e., columns of the fine division $\mathcal{C}_2$) we support the operation $\mathsf{colPerm}$ that returns a pointer to a column permutation structure (as described in part C).

Now we describe a space-efficient implementation of G_C in a recursive manner, starting from the level-2 nodes (leaves). Each level-2 node corresponds to a $\mathcal{C}_2$-column J and stores its width $|J|$ in $\mathcal{O}(\lg \lg n)$ bits followed by the index into the precomputed table described in part C. The methods $\mathsf{colPerm}$ and $\mathsf{colPerm}^{-1}$ simply read off the precomputed values from the corresponding table entry in constant time. The total space is thus $\lg s_\pi \cdot |J| + \mathcal{O}_\pi(\lg \lg n)$.

The data structure of level-1 nodes consists of three parts: (i) precomputed $\mathsf{rowRank}$ and $\mathsf{parentCell}$ queries, (ii) data needed to implement the $\mathsf{subColumn}$ method, and (iii) a concatenation of the data structures of its $\mathcal{C}_2$ subcolumns as bitstrings. Notice that each $\mathcal{C}_1$-column contains at most $q = 40 \lg^2 n / \sqrt{\lg n} \in \mathcal{O}(\lg^{3/2} n)$ columns of $\mathcal{C}_2$, and each $\mathcal{R}_1$-row contains at most q rows of $\mathcal{R}_2$, by Lemma 2.2(c).

Let us start by bounding the number of bits taken by the data structures of level-2 subcolumns (part (iii)). Since each level-2 column J takes $\lg s_\pi \cdot |J| + \mathcal{O}_\pi(\lg \lg n)$ bits, this adds to $\lg s_\pi \cdot w + \mathcal{O}_\pi(\lg^{3/2} n \cdot \lg \lg n) \subseteq \lg s_\pi \cdot w + \mathcal{O}_\pi(\lg^{7/4} n)$ where w is the width of the represented column. For (i), to store all $\mathsf{rowRank}(j, c)$ queries explicitly, we need to store a mapping $[q] \times [10c_\pi] \to [q] \times [10c_\pi]$ which requires $\mathcal{O}(\lg^{3/2} n \cdot 10c_\pi \cdot \lg(\lg n \cdot 10c_\pi)) \subseteq \mathcal{O}_\pi(\lg^{7/4} n)$ bits. Similarly, storing all $\mathsf{parentCell}(j, c)$ queries explicitly takes $\mathcal{O}_\pi(\lg^{7/4} n)$ bits.

In order to support the $\mathsf{subColumn}$ method, part (ii) of the node simply stores offsets to the beginning of each $\mathcal{C}_2$-subcolumn (part (iii)) in an array. Crucially, these offsets can be stored in a small number of bits since the concatenation of the data structures for level-2 subcolumns takes at most $\mathcal{O}(\lg s_\pi \cdot \lg^2 n)$ bits. Thus, each offset is stored in $\mathcal{O}_\pi(\lg \lg n)$ bits, for a total of $\mathcal{O}_\pi(\lg^{3/2} n \cdot \lg \lg n) \subseteq \mathcal{O}_\pi(\lg^{7/4} n)$. Altogether, this representation of a level-1 column fits into $\lg s_\pi \cdot w + \mathcal{O}_\pi(\lg^{7/4} n)$ bits.

The final data structure for the root node is essentially the same. It again consists of precomputed $\mathsf{rowRank}$ queries, offsets needed for access to subcolumns and a concatenation of the data structures of all $\mathcal{C}_1$ columns. (Recall that $\mathsf{parentCell}$ queries at the root are not needed.) To support $\mathsf{rowRank}(j, c)$ queries at the root, we need to store a mapping $[m_1] \times [10c_\pi] \to [m_1] \times [10c_\pi]$. This requires

$\mathcal{O}_\pi(n/\lg^2 n \cdot \lg n) \subseteq o(n)$ bits. The total space taken by the representations of level-1 columns is bounded by $\lg s_\pi \cdot n + \mathcal{O}_\pi(m_1 \cdot \lg^{7/4} n) \subseteq \lg s_\pi \cdot n + o(n)$. Finally, each offset needed for subColumn queries takes $\mathcal{O}(\lg n)$ bits for a total of $\mathcal{O}_\pi(n/\lg^2 n \cdot \lg n) \subseteq o(n)$. Therefore, the complete data structure representing gridded columns recursively takes $\lg s_\pi \cdot n + o(n)$ bits. We note that this component is the bottleneck of the entire data structure.

E. Recursive row structure G_R and row permutations rowPerm**.** This component is analogous to the column structure described in parts C and D. We similarly implement a method subRow(j) that gives a pointer to the j-th subrow of the root or a level-1 row node, colRank(j, c), that accesses the c-th non-zero cell of the j-th subrow, and returns (c', s), where c' is the identifier of the current cell in its column (among non-zero cells), and s is the relative rank of the column containing the cell. Similarly to part D, the method parentCell(j, c) returns a value c'', indicating that the c-th non-zero cell of the j-th subrow is contained in the c''-th non-zero cell of the current row.

The row permutations rowPerm are stored analogously to the column permutations. We write $(c, v) \leftarrow$ rowPerm(ℓ) or $\ell \leftarrow$ rowPerm$^{-1}(c, v)$ to indicate the mapping between c, the non-zero index of a cell within its row, a row offset ℓ, and a value v, the number of 1-entries that fall into the same cell c, below the queried entry. Again, we write rowPerm and rowPerm^{-1} as methods only for notational simplicity; these should be thought of as pointers to a global precomputed table from which the respective queries can be read out. To avoid duplication, we omit a more detailed description of this structure.

Similarly to C and D, the total space requirement is $n \lg s_\pi + o(n)$. The fact that we store both column and row permutations is what leads to the factor two in the leading term of the space bound.

Implementation of $\tau(i)$ and $\tau^{-1}(i)$ queries. We now describe the implementation of the main operations. The rank and unrank ($\tau(i)$ and $\tau^{-1}(i)$) queries are symmetric. We give the pseudocode for both in Figure 4 and a detailed textual description for the first.

Given i, i.e., a column index of the (unknown) permutation matrix M_τ, we find using rank and select over I_C the starting index $s_i \in [n]$ of the column in $\mathcal{C}_2$ where i falls, the index s of this column within $\mathcal{C}_2$ as well as the relative position (offset) s_3 within this column (clearly, $s_i + s_3 = i$). Note that if the columns of the division were of equal width, then we could compute these values by simple arithmetic, this, however is not guaranteed. The index 3 is meant to evoke the third (trivial) level of the hierarchy.

The column in $\mathcal{C}_2$ that contains i corresponds to the s-th leaf of the tree T_C. Using the tree interface, we find the rank s_2 of this leaf among its siblings, and the rank s_1 of its parent among its siblings. (In other words, the (fine) column containing i is the s-th in $\mathcal{C}_2$, and the s_2-th within the coarser column of $\mathcal{C}_1$ that contains it. In turn, the coarser column is the s_1-th in $\mathcal{C}_1$.)

We next navigate in the column structure G_C to the s_1-th child of the root and the s_2-th child of that node, using subColumn queries, i.e., to the $\mathcal{C}_1$ and $\mathcal{C}_2$ columns that contain the query point.

The $\mathcal{C}_2$-column is stored as a column permutation. Within this we can identify, using the column offset s_3, the cell index c_2 and within-cell vertical rank v. Recall that this means that the entry (τ_i, i) is in the c_2-th non-zero 2-cell in its $\mathcal{C}_2$-column and that this cell has v entries below (τ_i, i).

Using the rowRank and parentCell queries on the $\mathcal{C}_1$-column with the cell index c_2 we just found, and indicating that our $\mathcal{C}_2$-column is the s_2-th of its parent, we obtain three values. These are c_2', indicating that the 2-cell where the query point falls is the c_2'-th non-zero cell in its $\mathcal{R}_2$-row; r_2, meaning that this row is the r_2-th within its parent $\mathcal{R}_1$-row, and c_1, indicating that the 1-cell where the query point falls is the c_1-th non-zero cell in its $\mathcal{C}_1$-column. Using rowRank on the root, we can now determine that the 1-cell containing the query point is in the r_1-th $\mathcal{R}_1$-row.

Using the row indices r_1 and r_2 we navigate the row tree T_R to find the leaf corresponding to the $\mathcal{R}_2$-row containing the queried element. By querying its global leaf rank we identify the $\mathcal{R}_2$-row r

Input: i
Output: $\tau(i)$

$s \leftarrow I_C.\mathsf{rank}(i)$ ▷ *Find column rank*
$s_i \leftarrow I_C.\mathsf{select}(s)$ ▷ *Column start index*
$s_3 \leftarrow i - s_i$ ▷ *Offset within column*
$col \leftarrow T_C.\mathsf{leafSelect}(s)$ ▷ *Find relative column ranks*
$s_2 \leftarrow T_C.\mathsf{childRank}(col)$
$s_1 \leftarrow T_C.\mathsf{childRank}(T_C.\mathsf{parent}(col))$
$root \leftarrow G_C.\mathsf{root}$ ▷ *Navigate column structure*
$col_1 \leftarrow root.\mathsf{subColumn}(s_1)$
$col_2 \leftarrow col_1.\mathsf{subColumn}(s_2)$
$(c_2, v) \leftarrow col_2.\mathsf{colPerm}(s_3)$ ▷ *Cell id and rank-within-cell read from column permutation*
$(c'_2, r_2) \leftarrow col_1.\mathsf{rowRank}(s_2, c_2)$ ▷ *Cell id and relative row rank*
$c_1 \leftarrow col_1.\mathsf{parentCell}(s_2, c_2)$ ▷ *Parent cell id*
$(c'_1, r_1) \leftarrow root.\mathsf{rowRank}(s_1, c_1)$ ▷ *Find coarse row*
$root \leftarrow G_R.\mathsf{root}$ ▷ *Navigate row structure*
$row_1 \leftarrow root.\mathsf{subRow}(r_1)$
$row_2 \leftarrow row_1.\mathsf{subRow}(r_2)$
$r_3 \leftarrow row_2.\mathsf{rowPerm}^{-1}(c'_2, v)$ ▷ *Column offset read from row permutation*
$row \leftarrow T_R.\mathsf{child}(T_R.\mathsf{child}(T_R.\mathsf{root}, r_1), r_2)$ ▷ *Find fine row*
$r \leftarrow T_R.\mathsf{leafRank}(row)$
$r_i \leftarrow I_R.\mathsf{rank}(r)$ ▷ *Row start index*
return $r_i + r_3$

Input: i
Output: $\tau^{-1}(i)$

$r \leftarrow I_R.\mathsf{rank}(i)$ ▷ *Find row rank*
$r_i \leftarrow I_R.\mathsf{select}(r)$ ▷ *Row start index*
$r_3 \leftarrow i - r_i$ ▷ *Offset within row*
$row \leftarrow T_R.\mathsf{leafSelect}(r)$ ▷ *Find relative row ranks*
$r_2 \leftarrow T_R.\mathsf{childRank}(row)$
$r_1 \leftarrow T_R.\mathsf{childRank}(T_R.\mathsf{parent}(row))$
$root \leftarrow G_R.\mathsf{root}$ ▷ *Navigate row structure*
$row_1 \leftarrow root.\mathsf{subRow}(r_1)$
$row_2 \leftarrow row_1.\mathsf{subRow}(r_2)$
$(c'_2, v) \leftarrow row_2.\mathsf{rowPerm}(r_3)$ ▷ *Cell id and rank-within-cell read from row permutation*
$(c_2, s_2) \leftarrow row_1.\mathsf{colRank}(r_2, c'_2)$ ▷ *Cell id and relative column rank*
$c_1 \leftarrow row_1.\mathsf{parentCell}(r_2, c'_2)$ ▷ *Parent cell id*
$s_1 \leftarrow root.\mathsf{colRank}(r_1, c'_1)$ ▷ *Find coarse column*
$root \leftarrow G_C.\mathsf{root}$ ▷ *Navigate column structure*
$col_1 \leftarrow root.\mathsf{subColumn}(s_1)$
$col_2 \leftarrow col_1.\mathsf{subColumn}(r_2)$
$s_3 \leftarrow col_2.\mathsf{colPerm}^{-1}(c_2, v)$ ▷ *Row offset read from column permutation*
$col \leftarrow T_C.\mathsf{child}(T_C.\mathsf{child}(T_C.\mathsf{root}, s_1), s_2)$ ▷ *Find fine column*
$s \leftarrow T_C.\mathsf{leafRank}(col)$
$s_i \leftarrow I_C.\mathsf{rank}(s)$ ▷ *Column start index*
return $s_i + s_3$

Figure 4: Implementation of the rank and unrank queries.

where the query falls. Querying I_R we compute the exact vertical offset r_i of the row r (i.e., the starting coordinate of this row within the matrix).

It remains to compute the exact vertical offset within the row. For this we navigate G_R using r_1, r_2, to find the row permutation containing the query point. From this row permutation, using the vertical rank v and the horizontal index c'_2 of the 2-cell among non-zero cells, we obtain the exact vertical offset r_3 within the row.

Our global vertical offset (and the answer to the query $\tau(i)$) is now obtained by adding r_3 to r_i. As all steps take $\mathcal{O}(1)$ time, the overall cost of rank (as well as unrank) queries is $\mathcal{O}(1)$, independent of π. The bound of $2n \lg s_\pi + o(n)$ on the space usage follows from the description of the components.

Other operations. We briefly illustrate how other $\mathcal{O}(1)$-time operations can be implemented on top of the main data structure with little overhead. We omit describing several symmetric (rotated) variants of these operations, as they can be implemented very similarly.

We first discuss **range minimum**. As mentioned in § 2, with $2n + o(n)$ overhead, we can support range minimum queries over the permutation τ (and similarly, over τ^{-1}). We show that this can also be achieved with sublinear overhead. Most of the steps are similar to those of *rank* queries, we thus focus only on the necessary extensions.

On each $\mathcal{C}_2$-column we support queries $\mathsf{rm}(a, b)$ where a, b are relative column offsets within the $\mathcal{C}_2$-column and the query returns the offset $k \in [a, b]$ of the minimum entry within the interval. As such queries depend only on the permutation within the $\mathcal{C}_2$-column, they can be stored only once for all occurring (gridded) permutations, similarly to the $\mathsf{colPerm}(\mathsf{k})$ implementation described in part C. Storing the answer to each possible query for a column of width w requires $w^2 \cdot \lceil \lg w \rceil$ bits, for $w \in \mathcal{O}(\sqrt{\lg n})$. Again, as there are at most $\mathcal{O}(s_\pi^w \cdot w^{10c_\pi})$ such structures, the total space requirement is $o(n)$. Additionally, we build a range-minimum data structure over the minima of each $\mathcal{C}_2$-column. Given two $\mathcal{C}_2$-column indices i_a and i_b, the structure returns the index $i_k \in (i_a, i_b)$ of the $\mathcal{C}_2$ column with smallest minimum. The overhead of this structure is $2m_2 + o(m_2) \subseteq o(n)$.

Consider a query $\mathsf{rm}(a, b)$. We first identify the $\mathcal{C}_2$-columns where a and b fall. If a and b fall into the same $\mathcal{C}_2$-column, we can directly answer the query with a rm-query within the column. (The answer is an offset within the column, but we can easily translate this into a global column index.)

If a and b fall into different $\mathcal{C}_2$-columns, say i_a, i_b, then we perform four range-minimum queries: (1) in column i_a from the position of a to the end of the column, (2) in column i_b from the beginning of the column to b, (3) on the column-minima for columns with index between i_a and i_b, to identify column i_k with the smallest minimum, and finally (4) on column i_k to identify the actual minimum entry in this column. By using rank queries, we obtain the actual values of the entries returned in (1), (2), and (4), and return the index of the smallest among these.

The space requirement for the additional structures we created is $o(n)$. Note that other types of queries over columns (or rows) can be similarly handled.

We next describe **next smaller** queries. Here, given input i, we wish to return the smallest $j > i$ for which $\tau_j < \tau_i$.

First, given i, we identify the $\mathcal{C}_2$-column C and $\mathcal{R}_2$-row R where (τ_i, i) falls. We first verify whether there is a next smaller element in C; if there is, then that is the answer to the original query and we are done. As such queries depend only on the column permutation of a $\mathcal{C}_2$-column, we can precompute the answers to all possible queries and store them similarly to the $\mathsf{colPerm}(\mathsf{k})$ queries described in part C, in the table of column-permutations. We obtain the answer as an offset within C, and again, we can translate this into a global column index easily. The overhead is $o(n)$.

Now we issue a similar query in R. We obtain the answer as an offset within R, which we resolve into a global column index similarly to *unrank* queries. The obtained entry (τ_x, x), if exists, is a candidate for the overall next smaller.

Next, we look within the $\mathcal{C}_2$-columns that are to the right of the 2-cell of (τ_i, i), intersecting the 1-cell of (τ_i, i). We look for the leftmost element in these columns that is "above" the 2-cell of (τ_i, i) (recall that being above means having smaller τ-value). We identify such an element by its column index within the 1-cell, thus by $\mathcal{O}(\lg \lg n)$ bits. As the query depends only on the 2-cell containing the input, we can precompute the answers for all non-zero 2-cells, taking in total $\mathcal{O}(n/\sqrt{\lg n} \cdot 10c_\pi \cdot \lg \lg n) \subseteq o(n)$ bits. The obtained index can again be resolved to a global column index easily.

Symmetrically, we look for a candidate element within the $\mathcal{R}_2$-rows that are above the 2-cell of (τ_i, i), intersecting the 1-cell of (τ_i, i). We look for the leftmost element in these rows that is to the right of the 2-cell of (τ_i, i).

The only remaining location for the query answer is now fully to the right and above the 1-cell of (τ_i, i), namely the leftmost entry in this area. Since there are only $m_1 \cdot 10c_\pi$ 1-cells, we can precompute each such query, whose answer identifies the global column ($\mathcal{O}(\lg n)$ bits) that contains the solution. This takes $\mathcal{O}(n/\lg^2 n \cdot 10c_\pi \cdot \lg n) \subseteq o(n)$ bits.

We now have (up to) four candidate points as the answer to the original *next smaller* query. Each of these is feasible, so the overall answer is the leftmost among them.

Remark: We formulated our solution by decomposing the original query into queries over five areas: (1) the fine row of the input, (2) the fine column of the input, (3) the fine columns intersecting the coarse cell, to the right of the fine cell, (4) the fine rows intersecting the coarse cell, above the fine cell, and (5) the part of the matrix above and to the right of the coarse cell of the input. It is clear that these five areas together cover the area where the solution could lie. In certain applications it may be advantageous to decompose the query area into *disjoint* parts – in this way we can support various counting, and more general *semigroup queries*, where the answer is formed by composing the answers over disjoint components, as is common in computational geometry. It is not hard to make the query areas disjoint, by restricting part (2) to the area strictly above the fine cell (using the gridding of the permutation), and (3) to the area strictly above the coarse cell containing the input. For the sake of simplicity, we have avoided this technicality in our description. We note that similar issues arise and are handled in more detail in § 4.

Building the data structure. We briefly argue that the time to construct the data structure of Theorem 1.1 is linear. The main task is that of constructing the decomposition; by Lemma 2.2, this requires $\mathcal{O}(n \lg c_\pi) = \mathcal{O}(n \lg s_\pi)$ time. A key step, both in constructing the decomposition and in building the various components is that of *sorting the input*; after this step, points of the input can be navigated both horizontally and vertically. By [Opl24], sorting τ takes $\mathcal{O}(n \lg s_\pi)$ time. Constructing parts A and B clearly takes linear time. Storing the column and row permutations (parts C, E) again relies on efficient sorting. This takes $\mathcal{O}(w \lg s_\pi)$ time for a column (row) of width (height) w, adding to $\mathcal{O}(n \lg s_\pi)$ overall. Tabulating the permutations by size, isomorphism, and gridding, identifying and indexing the non-zero cells, and collecting the answers for each query take time linear in the representation, using straightforward (but slightly tedious) data structuring. The construction of the recursive row and column structures (parts D, E) and precomputation of rowRank, colRank, parentCell, subColumn, subRow can likewise be achieved by straightforward navigation of the input and its balanced decomposition structure; we omit a more detailed description.

4 Extension to geometric queries

In this section, we extend the data structure to geometric queries, proving Theorem 1.3. We will only describe the implementation of *rectangle range counting* queries, as the changes necessary to implement *range minimum* or other rectangle-based queries will be obvious.

We modify the original data structure of Theorem 1.1 in two major ways. First, we extend the row

and column permutations rowPerm and colPerm with precomputed geometric queries. Second, we build a new hierarchy of $\mathcal{O}(\lg\lg n)$ divisions with row and column sizes decreasing at a polynomial rate, alongside the original, two-level hierarchy.

Throughout the section, $(\mathcal{R}_1, \mathcal{C}_1)$ and $(\mathcal{R}_2, \mathcal{C}_2)$ refer to the two non-trivial divisions of the data structure in Theorem 1.1. We now define the sizes of divisions in the new hierarchy. Let $w_0, w_1, \ldots$ be a sequence where $w_0 = n$ and $w_i = \lfloor w_{i-1}^{5/6} \rfloor$, for $i \geq 1$, and let ℓ be the smallest integer such that $n/w_\ell \geq \lfloor n/\sqrt{\lg n} \rfloor$; observe that $\ell \in \Theta(\lg\lg n)$.

We compute ℓ levels of the balanced decomposition (Lemma 2.2) for parameters $1 < m_1 < \cdots < m_\ell < n$ where $m_i = \lfloor n/w_i \rfloor$ for $i < \ell$ and $m_\ell = \lfloor n/\sqrt{\lg n} \rfloor$. Let us denote the obtained divisions by $(\mathcal{R}'_1, \mathcal{C}'_1), \ldots, (\mathcal{R}'_\ell, \mathcal{C}'_\ell)$. We have made the specific choice for m_ℓ in order to guarantee that $(\mathcal{R}'_\ell, \mathcal{C}'_\ell)$ is exactly the same as $(\mathcal{R}_2, \mathcal{C}_2)$. (Since the algorithm of Lemma 2.2 is deterministic, it will always output on the same input, the same division of a given size.) We additionally let $(\mathcal{R}'_0, \mathcal{C}'_0)$ and $(\mathcal{R}'_{\ell+1}, \mathcal{C}'_{\ell+1})$ denote the two trivial divisions where $|\mathcal{R}'_0| = |\mathcal{C}'_0| = 1$ and $|\mathcal{R}'_{\ell+1}| = |\mathcal{C}'_{\ell+1}| = n$.

The crucial property of this hierarchy of divisions is that the number of subcolumns of each column is relatively small compared to its width: each column in $\mathcal{C}'_i$ contains at most $40 \cdot m_{i+1}/m_i \in \mathcal{O}(w_i^{1/6})$ columns of $\mathcal{C}'_{i+1}$ by Lemma 2.2(e), and the same holds analogously for rows.

We next describe the new or modified components of the data structure, referring to Figure 5 for an illustration.

C'. Extension of column and row permutations. We extend the column and row permutations with two additional queries. This will not affect the leading term in the space complexity, as this data is only stored in the precomputed tables, of sublinear total size.

Let J be a $\mathcal{C}_2$-column (alternatively $\mathcal{C}'_\ell$-column). We support new queries rectCount and cellOffset, defined as follows:

- rectCount$(i_1, i_2, c_1, j_1, c_2, j_2)$: values i_1, i_2 are column offsets within J, c_1, c_2 are (non-zero) cell indices, j_1, j_2 are relative row ranks within cells. The query returns the number of points in the column permutation, within the rectangle (including its boundaries) spanned by the i_1-th and i_2-th columns and the j_1-th row within the c_1-th cell and j_2-th row within the c_2-th cell.
- cellOffset(k, c): value k is a column offset within column J and c is a cell index. The query returns the number of points contained in the first k columns of J within the c-th cell.

We store the answers to all these queries precomputed in the tables. The total number of such queries is $\mathcal{O}(c_\pi^2(\sqrt{\lg n})^4) = \mathcal{O}_\pi(\lg^2 n)$ with each answer taking $\mathcal{O}(\lg\lg n)$ bits. The overall extra space needed in the precomputed tables is thus $\mathcal{O}(s_\pi^w \cdot w^{10c_\pi} \cdot c_\pi^2 w^4 \lg w) \subseteq o(n)$, where we used $w = \mathcal{O}(\sqrt{\lg n})$.

For row permutations, a query cellOffset(k, c) outputs the number of points in the c-th non-zero cell contained in the first k subrows and a query rectCount$(i_1, i_2, c_1, j_1, c_2, j_2)$ returns the number of points within the rectangle spanned by the i_1-th and i_2-th rows and the j_1-th column within the c_1-th cell and the j_2-th column within the c_2-th cell.

F. Dense column and row trees T'_C, T'_R. These trees are exact analogues of the trees T_C and T_R (part A), for the new, larger hierarchy of divisions, supporting the same navigational queries as T_C and T_R. Again, these take a linear number of bits in the total number of nodes, using the succinct tree implementation. Therefore, the overall space requirement is $\mathcal{O}(1 + \sum_{i=1}^{\ell} m_i) \subseteq \mathcal{O}(\ell \cdot n/\sqrt{\lg n}) \subseteq o(n)$ where the first inclusion holds since $m_i = \lfloor n/w_i \rfloor \leq \sqrt{n/\lg n}$ for every $i \in [\ell]$.

G. Dense recursive column and row structures G'_C, G'_R. Using the sequence of divisions, we build a data structure G'_C similar to G_C in part D of the original structure. (As G'_R is entirely symmetric,

we omit its description.) Again, the root corresponds to the interval $[n]$ (the single column of $\mathcal{C}'_0$), level-1 nodes correspond to the m_1 columns in $\mathcal{C}'_1$ and in general, level-i nodes correspond to the m_i columns in $\mathcal{C}'_i$ with the children of a node being its respective subcolumns within $\mathcal{C}'_{i+1}$. We build ℓ levels but we keep the leaves on the ℓ-th level without any stored information since we deal with the rows and columns of the division $(\mathcal{R}'_\ell, \mathcal{C}'_\ell) = (\mathcal{R}_2, \mathcal{C}_2)$ separately. We maintain the same recursive representation in memory where each node consists of some local information followed by offsets into the final part, a concatenation of data structures of its children. The tree is now significantly deeper, with $\ell \in \Theta(\lg \lg n)$ layers but importantly, its branching factor is smaller which allows for storing more information in each node.

For $k \in [\ell]$, a *k-cell* is a submatrix $M[I, J]$ where J is a $\mathcal{C}'_k$ column and I is a $\mathcal{R}'_k$-row. Let us consider a level-k node that represents a $\mathcal{C}'_k$-column J. We first describe a coordinate system we use to access the $(k+1)$-cells within J. On the horizontal axis, we index by an explicit offset i (rank) among the $\mathcal{C}'_{k+1}$-subcolumns of J. On the vertical axis, we describe the rows by a pair (c, j) where c is the rank of a non-zero k-cell within J and j is an explicit offset (rank) among the $\mathcal{R}'_{k+1}$-rows that intersect this k-cell. Therefore, a particular $(k+1)$-cell within I can be uniquely identified by a triple (i, c, j) and we say that it lies at coordinates (i, c, j).

The node implements three novel methods (see Figure 5): nonEmpty, cellRank and rectCount.

- nonEmpty(i, c, j) indicates if the $(k+1)$-cell at coordinates (i, c, j) is non-empty.
- cellRank(i, c, j) returns the number of non-empty $(k+1)$-cells above the $(k+1)$-cell at coordinates (i, c, j) in the same $\mathcal{C}'_{k+1}$-column, i.e., the number of non-empty $(k+1)$-cells at coordinates (i, c, j') for $j' < j$ and at coordinates (i, c', j'') for $c' < c$ and arbitrary j''. We can, moreover, input $j = 0$ ($j = \infty$ resp.) to get the number of non-empty $(k+1)$-cells in the i-th subcolumn contained in the first $c - 1$ (c resp.) non-empty k-cells.
- rectCount$(i_1, i_2, c_1, j_1, c_2, j_2)$ returns the number of points contained in the rectangle with opposite corners formed by the $(k+1)$-cells at coordinates (i_1, c_1, j_1) and (i_2, c_2, j_2) (inclusively). As in cellRank, we can also set either of j_1, j_2 to 0 or ∞. By setting $j_1 = 0$, we include all rows in the c_1-th non-empty k-cell whereas setting $j_1 = \infty$ includes only the rows strictly below the c_1-th k-cell. Similarly, setting $j_2 = 0$ excludes all rows in the c_2-th non-empty k-cell whereas $j_2 = \infty$ includes all of them in the query; note that we assume $c_1 < c_2$.

For simplicity, in all three cases we keep the signature of methods the same even for the root of G'_C. However, in that case the cell ranks (c, c_1 or c_2) must equal 1 since we consider the whole matrix M to be a single 0-cell.

We store all possible invocations of all three methods precomputed, available for constant-time retrieval. Additionally, each node supports navigational subColumn queries exactly as before.

Let us now bound the total size. First, we prove a crude bound that allows us to bound the number of bits needed for the subColumn offsets. By reverse induction on i, we prove that each level-i node fits into w_i^2 bits and in particular, the offsets needed for subColumn therein fit in $\mathcal{O}(\lg w_i)$ bits each. The leaves (level-ℓ nodes) do not store any information, so the claim holds vacuously. A general level-i node has $\mathcal{O}(w_i^{1/6})$ children, each taking up at most w_{i+1}^2 bits by induction. Plugging in $w_{i+1} \leq w_i^{5/6}$, we get that the concatenation of the children's representations takes at most $\mathcal{O}(w_i^{1/6} \cdot w_i^{10/6}) = \mathcal{O}(w_i^{11/6})$ bits which is less than $\frac{1}{3}w_i^2$ for large enough n since $w_i \geq \sqrt{\lg n}$. Thus, it suffices to reserve $2 \lg w_i$ bits for each subColumn offset for a total of $\mathcal{O}(w_i^{1/6} \lg w_i)$ bits which is, again, less than $\frac{1}{3}w_i^2$. Finally, the precomputed nonEmpty, cellRank and rectCount queries take $\mathcal{O}_\pi\big(\big(w_i^{1/6}\big)^4 \lg w_i\big) \in \mathcal{O}_\pi(w_i^{2/3} \lg w_i)$ bits of space, again less than $\frac{1}{3}w_i^2$ for large enough n. Summing together, we get that level-i nodes take at most w_i^2 bits in total, completing the induction.

Now, we do a more fine-grained global analysis of the overall space complexity. Observe that the final structure is simply a concatenation of the local data (subColumn offsets and precomputed

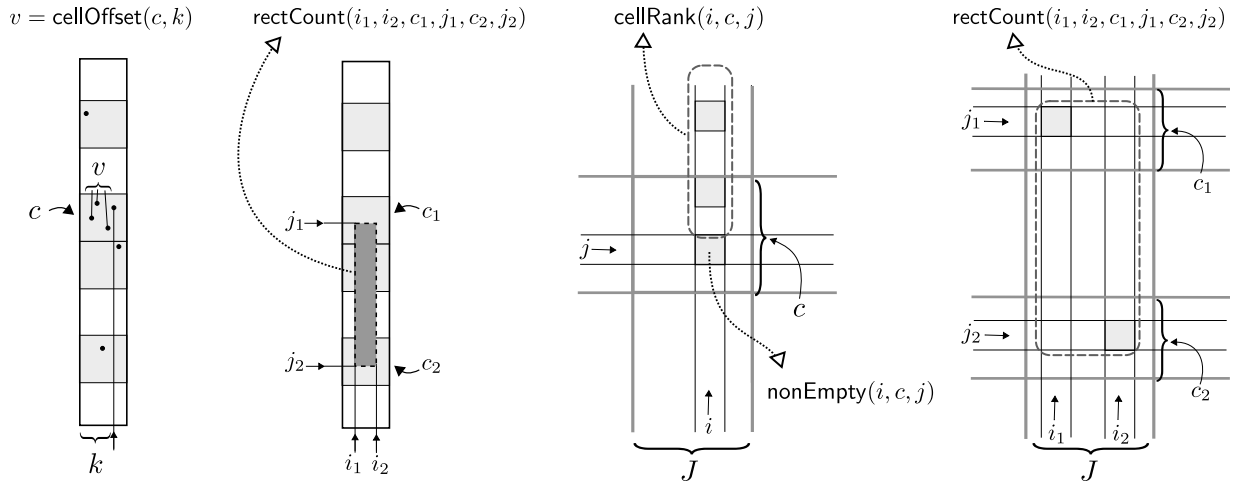

Figure 5: From left to right: (i) cellOffset query in $\mathcal{C}_2$-column: for absolute offset k and non-zero cell index c, we obtain the number of points v left of the query; (ii) rectCount query in $\mathcal{C}_2$-column: returns number of points in rectangle with corners identified by horizontal offsets i_1, i_2, and vertical offsets j_1, j_2 within non-zero cells c_1 and c_2; (iii) cellRank and nonEmpty queries for a column J (at arbitrary level), w.r. to cell in subcolumn i of J and subrow j within non-zero cell c of J; cellRank returns number of non-zero cells in marked area; (iv) rectCount query for a column J (at arbitrary level), i_1, i_2 now refer to subcolumns within J and j_1, j_2 refer to subrows within cells c_1, c_2 of J. Returns number of points in rectangle.

queries) for each node as encountered during an in-order traversal of the tree. Therefore, it suffices to bound the total space taken by the local data of each node. We do this by levels. On the i-th level, there are m_i nodes and each needs $\mathcal{O}(w_i^{1/6} \cdot \lg w_i)$ bits to store subColumn offsets and $\mathcal{O}_\pi(w_i^{2/3} \lg w_i)$ bits to store the precomputed queries. Together, this makes $\mathcal{O}_\pi(w_i^{2/3} \lg w_i)$ bits per level-i node for a total of $\mathcal{O}_\pi(m_i \cdot w_i^{2/3} \lg w_i) = \mathcal{O}_\pi(n \cdot w_i^{-1/3} \lg w_i)$ over the whole i-th level. Summing over all ℓ levels, we obtain an overall bound of $\mathcal{O}_\pi(n \cdot \sum_{i=1}^{\ell} w_i^{-1/3} \lg w_i)$ bits. Finally, plugging in $w_i \geq \sqrt{\lg n}$, we obtain the bound of $\mathcal{O}_\pi(n \cdot \ell \cdot \lg^{-1/6} n \cdot \lg \lg n) \subseteq o(n)$ bits.

4.1 Implementation of geometric queries

Let us now describe the implementation of a rectangular counting query, that asks for the number of points within a rectangle R. In the following, we generally use the four compass directions – north, west, south, east – referring to the four sides of the rectangle R – top, left, bottom, right.

Given a rectangle $R = [x^N, x^S] \times [x^W, x^E]$, we first identify the location of its corners within the hierarchy of divisions. This is done similarly as in the rank/unrank queries. We find using rank and select queries over I_R the indices r^S, r^N of the $\mathcal{R}'_\ell$-rows that contain x^S, x^N and the relative offsets $r^S_{\ell+1}, r^N_{\ell+1}$ within these rows. Next, we use T'_R to look up the relative ranks $r^S_1, \ldots, r^S_\ell$ and $r^N_1, \ldots, r^N_\ell$ where r^S_i (r^N_i resp.) is the relative rank of the $\mathcal{R}'_i$-row containing x^S (x^N resp.) within its parent $\mathcal{R}'_{i-1}$-row. Analogously, we obtain from I_C the indices s^W, s^E of the $\mathcal{C}'_\ell$-columns that contain x^W, x^E and the relative offsets $s^W_{\ell+1}, s^E_{\ell+1}$ within these columns. We use T'_C to look up the relative ranks $s^W_1, \ldots, s^W_\ell$ and $s^E_1, \ldots, s^E_\ell$ where s^W_i (s^E_i resp.) is the relative rank of the $\mathcal{C}'_i$-column containing x^W (x^E resp.) within its parent $\mathcal{C}'_{i-1}$-column. Additionally, we denote by v^α_i for $\alpha \in \{S, N, W, E\}$ the level-i node in G'_C or G'_R that contains the respective boundary of R. This phase performs constantly many queries to I_R, I_C and two traversals of T'_C, T'_R each, it thus takes $\mathcal{O}(\lg \lg n)$ time.

On a conceptual level, the answer is composed from precomputed rectCount counts similarly to

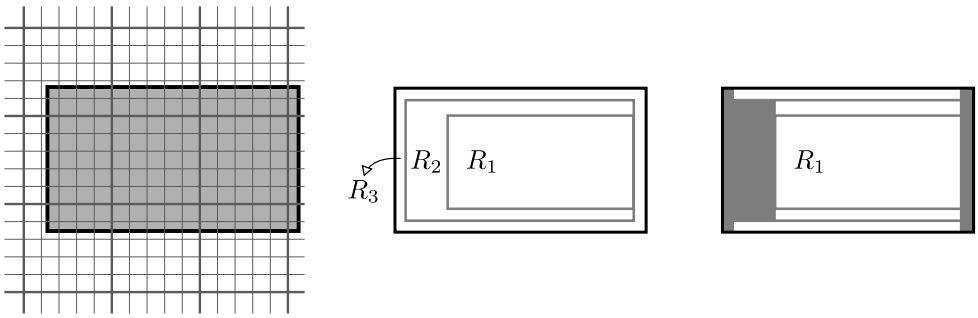

Figure 6: (i) Query rectangle R, with two levels of the decomposition shown. (ii) Decomposition of R into layers R_1, R_2, R_3. (iii) West- and east frame shown gray, north and south frame empty (above and below R_1 region).

the *next smaller* queries in § 3. We split the rectangle inductively into layers, where R_1 consists of all 1-cells fully contained within R, and for $i > 1$, the layer R_i consists of the i-cells fully contained within $R \setminus (R_1 \cup \cdots \cup R_{i-1})$. Note that there are $\ell + 1$ layers in total and we refer to R_i as the *i-layer* of R. Let $i_{\min}$ be the smallest integer such that there is an i-cell fully contained in R. By definition, $R_i = \emptyset$ for every $i < i_{\min}$. Observe that $R_{i_{\min}}$ is a rectangle while R_i for $i > i_{\min}$ is a set difference of two nested rectangles; see Figure 6.

Moreover for each $i > i_{\min}$, we split R_i into four disjoint parts $R_i^E, R_i^S, R_i^W, R_i^N$, one per each side of the rectangle R. We define R_i^W and R_i^E as the intersection of R_i with the leftmost and rightmost $\mathcal{C}'_{i-1}$-columns that intersect R. In other words, R_i^W (R_i^E resp.) consist of all i-cells that are fully contained in R and lie within the leftmost (rightmost resp.) $\mathcal{C}'_{i-1}$ column that intersects R. For R_i^S and R_i^N, we similarly restrict R_i to the bottommost and topmost $\mathcal{R}'_{i-1}$-row intersecting R. However, we additionally exclude the $(i-1)$-cells that contain the corners of R since these are already included in R_i^W and R_i^E. We refer to the union $R^E_{i_{\min}+1} \cup \cdots \cup R^E_\ell$ as the *east frame* of R. The west, south and north frames are defined analogously.

Counting points in $R_1 \cup \cdots \cup R_\ell$. Let us first assume, for the sake of simplicity, that all four corners of R fall into *different, empty* cells in the coarsest division $(\mathcal{R}'_1, \mathcal{C}'_1)$. In other words, each corner belongs to a different empty 1-cell and $i_{\min} = 1$. In this case, we use a single $\mathsf{rectCount}(s_1^W + 1, s_1^E - 1, 1, r_1^N + 1, 1, r_1^S - 1)$ query on the root of G'_C to get the number of points in R_1. This is because R_1 consists exactly of the 1-cells strictly between the s_1^W-th and s_1^E-th columns, and r_1^N-th and r_1^S-th rows. Next, we do two recursive traversals of G'_C and two of G'_R, one per each side of the rectangle R, to compute the contribution of east, west, north and south frames separately. Since these are symmetric, we only describe here the case of the east frame of R.

Fix $i > 1$, and let J be the $\mathcal{C}'_{i-1}$-column that contains the right boundary of R. Horizontally, the rectangle R_i^E spans exactly the first $s_i^E - 1$ columns of J since the boundary falls into the s_i^E-th subcolumn. We also know that R_i^E is vertically aligned with the non-empty $(i-1)$-cells in J since the corners of R fall into empty 1-cells. As we recurse into G'_C, we compute for each level-i the values $c_{i,1}^E$ and $c_{i,2}^E$ such that R_i^E spans vertically exactly the $c_{i,1}^E$-th up to the $c_{i,2}^E$-th $(i-1)$-cell. For $i = 2$, we set

$$c_{2,1}^E \leftarrow G'_C.\mathsf{root.cellRank}(s_1^E, 1, r_1^N) + 1$$
$$c_{2,2}^E \leftarrow G'_C.\mathsf{root.cellRank}(s_1^E, 1, r_1^S)$$

since we know the exact location of the northeast and southeast corners of R. For $i > 2$, we compute

$c^E_{i,1}, c^E_{i,2}$ from $c^E_{i-1,1}, c^E_{i-,2}$ at the node v^E_{i-1} of G'_C as follows:

$$c^E_{i,1} \leftarrow v^E_{i-1}.\mathsf{cellRank}(s^E_i, c^E_{i-1,1}, 0) + 1$$
$$c^E_{i,2} \leftarrow v^E_{i-1}.\mathsf{cellRank}(s^E_i, c^E_{i-1,2}, \infty).$$

This simply follows from the definition of $\mathsf{cellRank}$ using the fact that R^E_i is vertically aligned with the non-empty $(i-1)$-cells. Having these, we can immediately get the number of points in R^E_i as $v^E_{i-1}.\mathsf{rectCount}(1, s^E_i - 1, c^E_{i-1,1}, 0, c^E_{i-1,2}, \infty)$.

Repeating a symmetric procedure for the remaining three sides and adding the counts together, we obtain the desired number of points in $R_1 \cup \cdots \cup R_\ell$ in time $\mathcal{O}(\lg \lg n)$, since this amounts to four top-down traversals of G'_C and G'_R in total, with constantly many queries on each level.

Now, let us describe how to deal with the two additional assumptions, the first being the emptiness of the 1-cells containing the corners of R. We still assume that all four corners of R fall into different 1-cells (and, thus, $i_{\min} = 1$) but they can now be non-empty.

We generalize the procedure counting points in the east frame to this scenario. To that end, we additionally compute along the traversal for each level-i two flags $b^E_{i,1}$ and $b^E_{i,2}$ where $b^E_{i,1} = \texttt{true}$ ($b^E_{i,2} = \texttt{true}$ resp.) if and only if the southeast (northeast resp.) corner of R falls into a non-empty i-cell. We also specify the semantics (and computation) of $c^E_{i,1}$ and $c^E_{i,2}$ in this case. These now contain the minimum and maximum rank of non-empty $(i-1)$-cells within the respective $\mathcal{C}'_{i-1}$-column that intersects R. Observe that the semantics agree in the previous restricted case of empty 1-cells.

For $i = 1$, the values $b^E_{1,1}$ and $b^E_{1,2}$ are obtained as

$$b^E_{1,1} \leftarrow G'_C.\mathsf{root}.\mathsf{nonEmpty}(s^E_1, 1, r^N_1)$$
$$b^E_{1,2} \leftarrow G'_C.\mathsf{root}.\mathsf{nonEmpty}(s^E_1, 1, r^S_1).$$

For $i > 1$, we describe the computation of $b^E_{i,1}$ only, as the computation of $b^E_{i,2}$ is analogous. First, observe that if a corner of R falls into an empty $(i-1)$-cell then it necessarily falls into an empty i-cell. Otherwise, the southeast corner lies in the $c^E_{i-1,1}$-th non-empty $(i-1)$-cell. We set

$$b^E_{i,1} \leftarrow \begin{cases} \texttt{false} & \text{if } b^E_{i-1,1} = \texttt{false}, \text{ and} \\ v^E_{i-1}.\mathsf{nonEmpty}(s^E_i, c^E_{i-1,1}, r^N_i) & \text{otherwise,} \end{cases}$$

where we use the fact that the relative vertical offset of the northeast corner within its $(i-1)$-cell is exactly equal to r^N_i, that is, the relative rank of the $\mathcal{R}'_i$-row containing the north boundary of R within its $\mathcal{R}'_{i-1}$ parent. We similarly adapt the computation of $c^E_{i,1}$:

$$c^E_{i,1} \leftarrow \begin{cases} v^E_{i-1}.\mathsf{cellRank}(s^E_i, c^E_{i-1,1}, 0) + 1 & \text{if } b^E_{i-1,1} = \texttt{false}, \text{ and} \\ v^E_{i-1}.\mathsf{cellRank}(s^E_i, c^E_{i-1,1}, r^N_i) + 1 & \text{otherwise.} \end{cases}$$

Finally, we count the number of points in R^E_i as $v^E_{i-1}.\mathsf{rectCount}(1, s^E_i - 1, c^E_{i-1,1}, j_1, c^E_{i-1,2}, j_2)$ where

$$j_1 = \begin{cases} 0 & \text{if } b^E_{i-1,1} = \texttt{false}, \\ r^S_i + 1 & \text{otherwise.} \end{cases} \qquad j_2 = \begin{cases} \infty & \text{if } b^E_{i-1,2} = \texttt{false}, \\ r^N_i - 1 & \text{otherwise.} \end{cases}$$

The modified procedure still takes $\mathcal{O}(1)$ time on each level for a total of $\mathcal{O}(\lg \lg n)$ time.

The call to $\mathsf{rectCount}$ is symmetric for the west frame of R. However, we need to to modify the counting for the south and north frames to avoid overcounting. We only describe the difference for the north frame. We are computing the values $c^N_{i,1}, c^N_{i,2}$ and $b^N_{i,1}, b^N_{i,2}$ such that R^N_i intersects

horizontally exactly the $c^N_{i,1}$-th up to $c^N_{i,2}$-th non-empty i-cells in the corresponding $\mathcal{C}'_{i-1}$-row and $b^N_{i,1}, b^N_{i,2}$ capture whether the northheast and northwest corners of R fall into a non-empty i-cell. These values are computed analogously to the computation for the east frame described above. However, the final $\mathsf{rectCount}$ call changes to $v^N_{i-1}.\mathsf{rectCount}(1, r^N_i - 1, c^N_{i,1}, j_1, c^N_{i,2}, j_2)$ where

$$j_1 = \begin{cases} 0 & \text{if } b^N_{i-1,1} = \texttt{false}, \\ \infty & \text{otherwise.} \end{cases} \qquad j_2 = \begin{cases} \infty & \text{if } b^N_{i-1,2} = \texttt{false}, \\ 0 & \text{otherwise.} \end{cases}$$

This holds since the left and right boundaries of R^N_i are always aligned with $\mathcal{C}'_{i-1}$-columns and we only have to determine whether to include the $c^N_{i,1}$-th and $c^N_{i,2}$-th cell depending on whether they contain a corner of R.

Finally, we need to remove the assumption that all corners of R fall in different 1-cells. One possibility is that there exists i such that the four corners fall into the same $(i-1)$-cell but in four different i-cells. Observe that in that case we have $i_{\min} = i$ and the union $R_1 \cup \cdots \cup R_{i-1}$ of first $i-1$ layers is empty. We use the same top-down traversal, except we count points in R_i using a four-sided query at the node v^E_{i-1} and only start counting points in the four frames from level-i. However, we compute the values $c^E_{i,\cdot}$ and $b^E_{i,\cdot}$ (and analogously for the other three sides) starting at the root exactly as before.

The second option is that there are different i_1 and i_2 such that the right and left boundaries of R fall within the same $\mathcal{C}'_{i_1-1}$-column but in different $\mathcal{C}'_{i_1}$-columns, and the bottom and top boundaries of R fall within the same $\mathcal{R}'_{i_2-1}$-row but in different $\mathcal{R}'_{i_2}$-rows. Let us assume that $i_1 < i_2$, the other case being analogous. In this case, $i_{\min} = i_2$ and the union $R_1 \cup \cdots \cup R_{i_2-1}$ is empty since R cannot fully contain any i'-cell for $i' < i_2$. If $i_2 = \ell + 1$, then $R_1 \cup \cdots \cup R_\ell$ is empty and the query is fully resolved within the fine rows and columns in $\mathcal{O}(1)$ time as described below. Otherwise, we proceed as before with counting points in R_{i_2} at level-$(i_2 - 1)$ and starting counting points in the four frames from level-i_2.

Counting points in $R_{\ell+1}$. We consider two cases depending on whether all four corners of R lie in different ℓ-cells, or the rectangle R fits into a single $\mathcal{C}'_\ell$-column (the case of a single $\mathcal{R}'_\ell$-row is symmetric).

In the first case, we want to count the points in $R_{\ell+1}$ in each of the four frames separately using $\mathsf{rectCount}$ queries. We again describe the computation for the east frame only. We have previously computed the values $c^E_{\ell,1}, c^E_{\ell,2}$ and $b^E_{\ell,1}, b^E_{\ell,2}$ at level-$(\ell-1)$ in G'_C. If both $b^E_{\ell,1}$ and $b^E_{\ell,2}$ are $\texttt{false}$, we query $\mathsf{rectCount}(1, s^E_{\ell+1}, c^E_{\ell,1}, 0, c^E_{\ell,2}, \infty)$ on the finest column that contains the east boundary of R, using the column offset $s^E_{\ell+1}$ that was obtained from I_C at the beginning of the query. Otherwise, we cannot simply plug in the row offsets $r^N_{\ell+1}$ and $r^S_{\ell+1}$ for j_1 and j_2 since the finest columns are stored as permutations, potentially losing information about empty rows.

Nevertheless, the correct relative row offsets j_1, j_2 can be easily computed using $\mathsf{cellOffset}$ queries. If $b^E_{\ell,1} = \texttt{false}$, we have $j_1 = 0$ and otherwise, the desired row offset j_1 is given precisely by the query $\mathsf{cellOffset}(r^N_{\ell+1}, c^N_{\ell,2})$ on the $\mathcal{R}'_\ell$-row that contains the north boundary of R. Similarly, if $b^E_{\ell,1} = \texttt{false}$, we set $j_2 = \infty$ and otherwise, $j_2 = \mathsf{cellOffset}(r^S_{\ell+1}, c^S_{\ell,2})$ on the $\mathcal{R}'_\ell$-row that contains the south boundary of R.

It remains to deal with the case when, without loss of generality, R (and thus $R_{\ell+1}$) fits into a single $\mathcal{C}'_\ell$-column. However, in that case the cell ranks $c^N_{\ell,1}$ and $c^N_{\ell,2}$ must be equal because the northwest and northeast corners fall into the same ℓ-cell. The same holds for $c^S_{\ell,1}$ and $c^S_{\ell,2}$. Therefore, we first compute the relative row offset j_1 of the north boundary by querying $\mathsf{cellOffset}(r^N_{\ell+1}, c^N_{\ell,2})$ on the $\mathcal{R}'_\ell$-row containing the north boundary. Similarly, the relative row offset j_2 is obtained by querying $\mathsf{cellOffset}(r^S_{\ell+1}, c^S_{\ell,2})$ on the $\mathcal{R}'_\ell$-row with the south boundary. The number of points in $R_{\ell+1}$ is then given by the query $\mathsf{rectCount}(s^W_{\ell+1}, s^E_{\ell+1}, c^E_{\ell,1}, j_1, c^E_{\ell,2}, j_2)$.

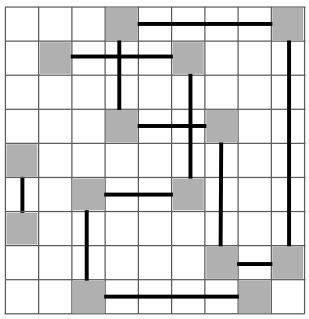

Figure 7: Three components of a division, with non-zero cells shaded gray, edges indicated with thick lines.

5 Bounded treewidth case

In this section, we prove Theorem 1.2. A key concept we use, in the context of balanced decompositions, is that of a *component*.

Given a division, we consider the graph whose vertices are the non-zero cells of the division and two vertices are connected by an edge exactly if the two corresponding cells are in the same column or row of the division. We refer to this graph as the graph of the division, and to its connected components as the *components of the division*, see Figure 7. The key observation is that if the treewidth of τ is bounded, then we can find a balanced decomposition similar to the one in Lemma 2.2, with the additional property that components of the division are of bounded size. This is captured in the following result, with proof in § 7.

Lemma 5.1. *Let σ be a permutation of length n and treewidth t.*

For integer parameters $1 < m_1 < m_2 < \cdots < m_\ell < n$, there exist divisions $(\mathcal{R}_1, \mathcal{C}_1), \ldots, (\mathcal{R}_\ell, \mathcal{C}_\ell)$ of M such that:

(a) The division $(\mathcal{R}_i, \mathcal{C}_i)$ is a coarsening of $(\mathcal{R}_{i+1}, \mathcal{C}_{i+1})$ for each $i \in [\ell - 1]$.

(b) $M(\mathcal{R}_i, \mathcal{C}_i)$ has exactly m_i rows and columns.

(c) Each row in $\mathcal{R}_i$ and each column in $\mathcal{C}_i$ has height, resp., width at most $\mathcal{O}(t \cdot n/m_i)$.

(d) Each component in $(\mathcal{R}_i, \mathcal{C}_i)$ spans at most $\mathcal{O}(t)$ rows and columns.

(e) Each row in $\mathcal{R}_i$ and each column in $\mathcal{C}_i$ is obtained from merging at most $\mathcal{O}(t \cdot m_{i+1}/m_i)$ rows of $\mathcal{R}_{i+1}$, resp., columns of $\mathcal{C}_{i+1}$, for $i \in [\ell - 1]$.

Moreover, the divisions $(\mathcal{R}_1, \mathcal{C}_1), \ldots, (\mathcal{R}_\ell, \mathcal{C}_\ell)$ can be computed in time $\mathcal{O}_t(n + \sum_{i=1}^{\ell} m_i)$.

Using Lemma 5.1, the data structure for storing a permutation simplifies. Instead of the recursive row- and column- structures we stored earlier, we now build a single recursive structure based on the components. Assuming bounded treewidth, components are still sufficiently small to be stored in a global, precomputed table. A component-permutation allows converting, for the entries in the component, absolute column-information to row-information and vice versa.

We remark in passing that the notion of component and the decomposition in Lemma 5.1 relate to the concept of *component twin-width* [BKRT22]. Loosely speaking, we are computing a *balanced* component twin-width decomposition, analogously to how Lemma 2.2 achieves a balanced twin-width decomposition.

Again, we consider divisions $(\mathcal{R}_1, \mathcal{C}_1)$ and $(\mathcal{R}_2, \mathcal{C}_2)$ of the input, with the same parameters as in § 3, namely $m_1 = n/\lg^2 n$ and $m_2 = n/\sqrt{\lg n}$. We invoke Lemma 5.1, observing the new property that

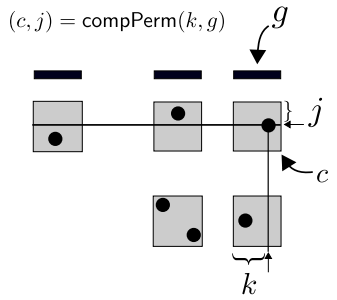

Figure 8: Given the (gridded) permutation of a component, we can identify, from the horizontal offset k within gridding column g the non-zero cell c where the query falls and the absolute vertical offset j within its row. Gray boxes indicate cells of the component, thick lines above indicate gridding columns. Dots other than the query are to be interpreted schematically as possibly multiple points.

the components of both divisions are of size $O(t)$, where t is the treewidth of τ. In the following, assume that t is constant.

The following parts of the data structure differ from the one in § 3.

C'. Component permutations (replacing both column and row permutations). Each component of $(\mathcal{R}_2, \mathcal{C}_2)$ is stored as a gridded permutation. By *gridded permutation* we now mean the permutation of points in a component, together with the horizontal and vertical partitioning, according to the different cells $(\mathcal{R}_2, \mathcal{C}_2)$ where they fall. (Contrast this with the column permutations in § 3 that are only vertically split into cells, and row permutations that are only horizontally split.)

The representation supports the query $\mathsf{compPerm}(k, g) \to (c, j)$. The interpretation is that the entry in the k-th column within the g-th column of the gridding falls into the c-th non-zero cell with relative row offset j; see Figure 8. It also supports the query $\mathsf{compPerm}^{-1}(k, g) \to (c, j)$, where the entry in the k-th row within the g-th row of the gridding falls into the c-th non-zero cell of the component with relative column offset j. Non-zero cells within a component are indexed in an arbitrary canonical way, say first left to right, then top to bottom.

The domain of both queries is $\mathcal{O}(\sqrt{\lg n}) \times \mathcal{O}(t)$ and their range is $\mathcal{O}(t) \times \mathcal{O}(\sqrt{\lg n})$. Thus, the precomputed answers for all such queries for a component gridded permutation take $\mathcal{O}(\sqrt{\lg n} \cdot t \cdot (\lg \lg n + \lg t))$ bits.

Suppose that the total width of a component is w. Using an upper bound of $\mathcal{O}(\sqrt{\lg n})$ on the column width, and the fact that a component consists of at most $O(t)$ cells, the number of possible gridded permutations that arise in a component is at most $s_\pi^w \cdot \sqrt{\lg n}^{\mathcal{O}(t)}$, which means that an index to the component-permutation can be implemented in $w \lg s_\pi + \mathcal{O}(\lg \lg n)$ bits, which, over all components adds up to $n \lg s_\pi + o(n)$ bits. (Notice that no two components can overlap in a column of $\mathcal{C}_2$, as otherwise they would merge, and that the number of components is at most $m_2 = n/\sqrt{\lg n}$.) In case of a supermultiplicative subclass $\mathcal{C}$ of π-avoiding permutations the total bound changes accordingly to $\lg |\mathcal{C}_n| + o(n)$ bits.

Storing all the precomputed queries for at most $s_\pi^w \cdot \sqrt{\lg n}^{\mathcal{O}(t)}$ distinct gridded permutations requires $o(n)$ bits overall, by a similar calculation as earlier.

D'. Recursive component structure G (replacing both G_C and G_R). This is a three-level tree similar to G_C and G_R.

We already discussed the storage of components of $(\mathcal{R}_2, \mathcal{C}_2)$. We similarly define components of $(\mathcal{R}_1, \mathcal{C}_1)$. We say that a component of $(\mathcal{R}_2, \mathcal{C}_2)$ is a subcomponent of a component of $(\mathcal{R}_1, \mathcal{C}_1)$, if they intersect in any entry of the input. Notice that this immediately means that the larger

component fully contains the smaller (if the two components overlap on some entry, then every cell of the smaller must be contained in some cell of the larger).

The root of G corresponds to the entire input, and level-1, resp., level-2 nodes correspond to $(\mathcal{R}_1, \mathcal{C}_1)$, resp., $(\mathcal{R}_2, \mathcal{C}_2)$ components, by containment.

For level-1 nodes, the data structure supports the query $\mathsf{subComponent}(k, g) \to (g', p)$ where the k-th subcolumn within the g-th column of the gridding falls into the g'-th column of the gridding of subcomponent p. Similarly, $\mathsf{subComponent}^{-1}(k, g) \to (g', p)$ means that the k-th subrow within the g-th row of the gridding falls into the g'-th row of the gridding of subcomponent p. For the root the queries are similar, but with the second parameter g missing (since the entire matrix can be viewed as a single cell/component where we need not provide gridding column or row).

Additionally, the root supports $\mathsf{rowRank}(k, c) \to r$ and $\mathsf{colRank}(k, c) \to r$ implemented exactly as in § 3. Level-1 nodes (components of $(\mathcal{R}_1, \mathcal{C}_1)$ support the query $\mathsf{rowRank}(k, c) \to (c', r)$ and $\mathsf{colRank}(k, c) \to (c', r)$ where c and c' now refer to the non-zero index of a cell in its component, but otherwise they are identical to the methods in § 3.

Level-2 nodes store a pointer to the component permutation structure that can be queried, as discussed in part C'.

The total space usage can be bounded almost identically to G_C and G_R in § 3, with components playing the role of columns (or rows), we therefore omit the detailed calculation. Notice that the number of components cannot be larger than the number of rows or columns in any division. In fact, now the number of non-zero cells in columns (and rows) is also at most $O(t)$.

The saving of a factor two compared to Theorem 1.1 comes from the fact that we need not separately store row and column structures, but the single component structure suffices. The individual methods that we had to implement in duplicate for rows/columns all have sublinear contribution to the total space, the bottleneck is the table of component-permutations and its indexing, which is stored only once.

Implementation of the queries. In case of **rank queries** (Figure 9), as before, we start by obtaining column indices and offsets from I_C and T_C. Then we navigate the component structure G using $\mathsf{subComponent}$ queries, to obtain the component of $(\mathcal{R}_1, \mathcal{C}_1)$ and $(\mathcal{R}_2, \mathcal{C}_2)$ that contain the query point. Using the component of the fine division, we get via $\mathsf{compPerm}$ the corresponding permutation, in which we resolve the query, obtaining the index of the non-zero cell in which the query falls and its vertical offset within its cell.

Tracing up through G, we obtain the row ranks at the coarse level from the coarse level component, and at the root, using $\mathsf{rowRank}$ queries.

The endgame is exactly as in Theorem 1.1; having obtained the row offset within the cell and relative row ranks within $\mathcal{R}_1$ and $\mathcal{R}_2$, we navigate T_R and I_R to compute the absolute row index which is the final output.

The **unrank queries** can now be implemented similarly. We first obtain row indices and offsets from I_R and T_R. Then we navigate the component structure to obtain the components containing the query point, this time with $\mathsf{subComponent}^{-1}$ queries. Then we identify the non-zero cell in which the query falls, and its horizontal offset via $\mathsf{compPerm}^{-1}$ queries. Going to the higher levels of G, we find via $\mathsf{colRank}$ queries the column ranks at the coarse- and top levels. Finally, we navigate T_C and I_C to compute the absolute column index, which is the output of the query.

Remark. Intuitively, the advantage of the component-based approach, compared to the earlier row/column-scheme (§ 3), is that components fully span both rows and columns, thus, collapsing them to a permutation loses no information. To appreciate this difference, recall that in a column permutation, having identified a point in a cell c with a $\mathsf{colPerm}$ query, we could only extract the relative rank v of the point *within its cell* (number of points below it) – see Figure 3 (left). This is

Input: i
Output: $\tau(i)$

$s \leftarrow I_C.\mathsf{rank}(i)$ ▷ *Find column rank*
$s_i \leftarrow I_C.\mathsf{select}(s)$ ▷ *Column start index*
$s_3 \leftarrow i - s_i$ ▷ *Offset within column*
$col \leftarrow T_C.\mathsf{leafSelect}(s)$ ▷ *Find relative column ranks*
$s_2 \leftarrow T_C.\mathsf{childRank}(col)$
$s_1 \leftarrow T_C.\mathsf{childRank}(T_C.\mathsf{parent}(col))$
$root \leftarrow G.\mathsf{root}$ ▷ *Navigate component structure*
$(g_1, comp_1) \leftarrow root.\mathsf{subComponent}(s_1)$
$(g_2, comp_2) \leftarrow comp_1.\mathsf{subComponent}(s_2, g_1)$
$(c'_2, r_3) \leftarrow comp_2.\mathsf{compPerm}(s_3, g_2)$ ▷ *Cell id and offset-within-cell read from component permutation*
$(c'_1, r_2) \leftarrow comp_1.\mathsf{rowRank}(s_2, c'_2)$ ▷ *Parent cell id and relative row ranks*
$r_1 \leftarrow root.\mathsf{rowRank}(s_1, c'_1)$
$row \leftarrow T_R.\mathsf{child}(T_R.\mathsf{child}(T_R.\mathsf{root}, r_1), r_2)$ ▷ *Find row rank*
$r \leftarrow T_R.\mathsf{leafRank}(row)$
$r_i \leftarrow I_R.\mathsf{rank}(r)$ ▷ *Row start index*
return $r_i + r_3$

Figure 9: Implementation of the rank query for bounded treewidth permutations.

because we did not have access to other non-zero cells in the same row as c that may also contain points below the query, so we could not directly compute the rank of the query point within the entire row. (Put differently, this is exactly why we needed to access the row permutation as well.) In contrast, now *all* non-zero cells in the same row or same column as c are in the same component, and thus, all the necessary information to compute the exact vertical or horizontal rank is locally available.

We note that combining the design of this section with elements of the $O(\lg \lg n)$-level hierarchy of § 4, we could support similar geometric queries with sublinear space overhead, i.e., succinctly (assuming bounded treewidth); we omit the details of this extension.

6 Proof of the structural lemma

In this section we prove the balanced decomposition result (Lemma 2.2).

We first describe an algorithm that requires knowledge of the pattern π and in particular its Füredi-Hajnal limit c_π. Afterwards, we describe the generalization to unknown π borrowing ideas from the optimal pattern-avoiding-sorting algorithm [Opl24].

Prior knowledge of c_π. Let $d = 5c_\pi$, $\delta = 20$. We say that a row I in $M(\mathcal{R}, \mathcal{C})$ is *tall* if $|I| > \delta n/|\mathcal{R}|$, and a column J is *wide* if $|J| > \delta n/|\mathcal{C}|$.

We first give a high-level description of the algorithm and only afterwards we describe its efficient implementation in full detail. We additionally define $m_0 = 1$ and $m_{\ell+1} = n$. The algorithm iteratively computes a sequence of divisions $(\mathcal{R}'_n, \mathcal{C}'_n), \ldots, (\mathcal{R}'_1, \mathcal{C}'_1)$ such that

(i) The division $(\mathcal{R}'_i, \mathcal{C}'_i)$ is a coarsening of $(\mathcal{R}'_{i+1}, \mathcal{C}'_{i+1})$.

(ii) The division $(\mathcal{R}'_i, \mathcal{C}'_i)$ has exactly i rows and columns.

(iii) Each row (resp. column) of $M(\mathcal{R}'_i, \mathcal{C}'_i)$ has height (resp. width) at most $2\delta \cdot n/i$.

(iv) Each row and column in $M(\mathcal{R}'_i, \mathcal{C}'_i)$ is obtained from merging at most $2\delta \cdot m_{j+1}/m_j$ rows or columns of $M(\mathcal{R}'_{m_{j+1}}, \mathcal{C}'_{m_{j+1}})$, where j is the largest integer such that $m_j \leq i$.

(v) Each row and column in $M(\mathcal{R}'_i, \mathcal{C}'_i)$ contains at most d non-zero cells.

Observe that we get the desired output by setting $(\mathcal{R}_i, \mathcal{C}_i) = (\mathcal{R}'_{m_i}, \mathcal{C}'_{m_i})$. Thus, from now on we focus on computing the sequence $(\mathcal{R}'_n, \mathcal{C}'_n), \ldots, (\mathcal{R}'_1, \mathcal{C}'_1)$.

Initially, we set $(\mathcal{R}'_n, \mathcal{C}'_n)$ to be the trivial (finest) division where each row (resp. column) contains exactly one row (resp. column) of the input matrix M. We then run $\ell + 1$ phases with the j-th phase generating divisions $(\mathcal{R}'_i, \mathcal{C}'_i)$ for $i \in \{m_j, \ldots, m_{j+1} - 1\}$. We say that a row in the j-th phase is *dense* if it contains more than $\delta \cdot m_{j+1}/m_j$ rows of the division $(\mathcal{R}'_{m_{j+1}}, \mathcal{C}'_{m_{j+1}})$, a dense column is defined analogously.

The algorithm generates $(\mathcal{R}'_i, \mathcal{C}'_i)$ from $(\mathcal{R}'_{i+1}, \mathcal{C}'_{i+1})$ in the following way. First, it finds an arbitrary pair of adjacent rows $R_1, R_2 \in \mathcal{R}'_{i+1}$ such that R_1 and R_2 are both neither tall nor dense, and the row $R_1 \cup R_2$ contains at most d non-empty cells with respect to $\mathcal{C}'_{i+1}$. Similarly, it finds an arbitrary pair of adjacent columns $C_1, C_2 \in \mathcal{C}'_{i+1}$ such that C_1 and C_2 are both neither wide nor dense, and the column $C_1 \cup C_2$ contains at most d non-empty cells with respect to $\mathcal{R}'_{i+1}$. Afterwards, the division $(\mathcal{R}'_i, \mathcal{C}'_i)$ is obtained from $(\mathcal{R}'_{i+1}, \mathcal{C}'_{i+1})$ by replacing R_1, R_2 and C_1, C_2 with their respective unions.

Correctness. For now, let us assume that the algorithm is able to generate all n divisions. In that case, we show that $(\mathcal{R}'_i, \mathcal{C}'_i)$ satisfies all the properties (i)–(v) by reverse induction on i. Clearly, all of them hold for the initial division $(\mathcal{R}'_n, \mathcal{C}'_n)$. Now, assume that $i < n$. The properties (i) and (ii) follow directly from the algorithm. Towards showing (iii), assume that there exists a row R in $\mathcal{R}'_i$ taller than $2\delta \cdot n/i$. It must have been obtained by merging two non-tall rows in some step k for $k > i$, i.e., each has height at most $\delta \cdot n/k$. But then the height of R is at most $2\delta \cdot n/k \leq 2\delta \cdot n/i$. Symmetrically, (iii) holds also for columns. The situation of (iv) is similar. Any row R in $(\mathcal{R}'_i, \mathcal{C}'_i)$ that contains more than one row (column) of the division $(\mathcal{R}'_{m_{j+1}}, \mathcal{C}'_{m_{j+1}})$ was obtained by merging two non-dense rows in kth step where $j + 1 \geq k > i$. A non-dense row in kth step contains at most $\delta \cdot m_{j+1}/m_j$ rows of $(\mathcal{R}'_{m_{j+1}}, \mathcal{C}'_{m_{j+1}})$ and thus, we get that R contains at most $2\delta \cdot m_{j+1}/m_j$ rows of $\mathcal{R}'_{m_{j+1}}$. Analogous arguments show the same inequality for columns. Finally, notice that merging pairs of adjacent rows cannot increase the number of non-empty cells in any fixed column and vice versa. Moreover, the algorithm merges adjacent rows (columns) only if their union contains at most d non-empty cells. Therefore, the algorithm never creates a row or column with more than d non-empty cells and (v) holds.

Assume for a contradiction that the algorithm was not able to construct the division $(\mathcal{R}'_i, \mathcal{C}'_i)$ from $(\mathcal{R}'_{i+1}, \mathcal{C}'_{i+1})$ for some $i \in [n-1]$. Let j be the index of the corresponding phase, i.e., the largest integer such that $m_j \leq i$. We only consider the case when there was no suitable pair of adjacent rows R_1 and R_2 as the case of columns is analogous.

Let us denote by T and D the number of tall rows and dense rows, respectively. First observe that $T < \frac{i+1}{\delta}$ since each tall row has size strictly larger than $\delta \cdot \frac{n}{i+1}$ and thus, $T \leq \frac{i}{\delta}$ as δ is an integer. Moreover, the number of dense rows can be bounded as

$$D < \frac{m_{j+1}}{\delta \frac{m_{j+1}}{m_j}} = \frac{m_j}{\delta} \leq \frac{i+1}{\delta},$$

where the first inequality follows since each dense row contains strictly more than $\delta \frac{m_{j+1}}{m_j}$ columns out of the m_{j+1} columns of $\mathcal{C}'_{m_{j+1}}$, and the last inequality is because $i + 1 \geq m_j$ in the jth phase by definition. By the same argument as before, we get the non-strict inequality $D \leq \frac{i}{\delta}$ since δ is an integer.

The number of adjacent pairs of rows that are both non-tall and non-dense is at least $\lfloor (i+1)/2 \rfloor - T - D \geq i/2 - T - D$. Since the algorithm halted unsuccessfully, each of these adjacent pairs must

Input: π-avoiding permutation σ, parameters $1 < m_1 < m_2 < \cdots < m_\ell < n$

$d \leftarrow 1, j \leftarrow \ell$
$(\mathcal{R}, \mathcal{C}) \leftarrow$ the trivial division of M_σ into n rows and columns of size 1
for $i \leftarrow n-1, \ldots, 1$ **do**
  if Q_1 or Q_2 is empty **then** ▷ *Triggers immediately in the first iteration.*
    $d \leftarrow 2d$
    for all $T \in \mathcal{R} \cup \mathcal{C}$ **do** QueueForMerging(T)
  MergeRows(Q_1.pop()) ▷ *Merge a pair of adjacent rows from Q_1.*
  MergeCols(Q_2.pop()) ▷ *Merge a pair of adjacent columns from Q_2.*
  for all $T \in B[i]$ **do** ▷ *Queue non-tall rows, non-wide columns for merging.*
    QueueForMerging(T.prev)
    QueueForMerging(T)
  if $i = m_j$ **then**
    $(\mathcal{R}_j, \mathcal{C}_j) \leftarrow (\mathcal{R}, \mathcal{C})$ ▷ *Copy current division.*
    for all $T \in \mathcal{R} \cup \mathcal{C}$ **do** T.density $\leftarrow 1$ ▷ *Reset density.*
    for all $T \in \mathcal{R} \cup \mathcal{C}$ **do** QueueForMerging(T) ▷ *Add to queue non-dense.*
    $j \leftarrow j - 1$
procedure QueueForMerging(T)
  ▷ *Checks if T can be merged with T.next and if yes, adds them to Q_1 or Q_2.* ◁
  if neither of T, T.next is wide/tall, dense or in the queue **then**
    if T.cellsWithNext $\leq d$ **then**
      T.inQueue $\leftarrow$ `true`, T.next.inQueue $\leftarrow$ `true`
      if T is a row **then** Q_1.push(T, T.next) ▷ *Push to the column queue.*
      else Q_2.push(T, T.next) ▷ *Push to the row queue.*

Figure 10: Algorithm for computing the structural decomposition.

together contain strictly more than d non-empty cells. Therefore, the number of non-empty cells in $M(\mathcal{R}'_{i+1}, \mathcal{C}'_{i+1})$ is at least

$$\left(\frac{i}{2} - \frac{i}{\delta} - \frac{i}{\delta}\right) \cdot d \geq \left(\frac{1}{2} - \frac{1}{20} - \frac{1}{20}\right) \cdot i \cdot d \geq \frac{2}{5} \cdot d \cdot \frac{i+1}{2} = c_\pi \cdot (i+1).$$

We obtain that $M(\mathcal{R}'_{i+1}, \mathcal{C}'_{i+1})$ contains π by Lemma 2.1 and thus also M contains π, a contradiction.

Without knowledge of c_π. It turns out that the algorithm is easily adapted to the case when c_π is not known in advance. In this case, we initially set $d = 1$ and double this parameter whenever there is no pair of adjacent rows or columns available for merging. It is clear that the algorithm always generates a sequence of divisions $(\mathcal{R}'_n, \mathcal{C}'_n), \ldots, (\mathcal{R}'_1, \mathcal{C}'_1)$, and it remains to observe that they have the desired properties.

The properties (i)–(iv) are independent of the value of d and thus, hold for $(\mathcal{R}'_n, \mathcal{C}'_n), \ldots, (\mathcal{R}'_1, \mathcal{C}'_1)$ regardless of the value of d. Property (v) implies that each row and column in every division contains at most $d_{\max}$ non-empty cells where $d_{\max}$ is the maximum (final) value of d throughout the algorithm. We claim that $d_{\max} \leq 10c_\pi$. Observe that otherwise the algorithm was unable to generate the sequence of divisions for $d = d_{\max}/2 \geq 5c_\pi$ and we reach a contradiction exactly as in the case when c_π was known.

Efficient implementation. We now proceed to describe an efficient implementation of the algorithm. The current division $(\mathcal{R}'_i, \mathcal{C}'_i)$ is stored in a structure where each row, column and non-empty cell is

stored in a separate record that are interlinked with pointers. The operation of merging a pair of adjacent rows or columns can then be implemented locally in time proportional to the number of non-empty cells involved. We postpone the full details of this structure for now and focus first on the high-level overview.

Central to our implementation is the ability to detect mergeable pairs of rows and columns. To that end, the algorithm maintains two queues Q_1 and Q_2 that store pointers to mergeable pairs of rows and columns, respectively. It is important to observe that if two adjacent rows (or columns) can be merged within the division $(\mathcal{R}'_i, \mathcal{C}'_i)$ then they can also be merged within any later division $(\mathcal{R}'_j, \mathcal{C}'_j)$ for $j > i$, assuming they still exist. The algorithm will maintain the following two invariants pertaining to the queues Q_1 and Q_2:

(Q1) each row appears at most once in Q_1 and each column appears at most once in Q_2, and

(Q2) for any pair of mergeable rows R_1 and R_2, at least one of R_1, R_2 is contained in Q_1, and analogously for mergeable columns and Q_2.

As an immediate corollary, we get that Q_1 is empty if and only if there is no mergeable pair of rows and analogously for Q_2 and columns. Therefore in each step, the algorithm checks whether both Q_1 and Q_2 are non-empty. If yes, it pops a pair of mergeable rows from Q_1 and a pair of mergeable columns from Q_2 and performs the respective merge operation. Otherwise, it doubles the parameter d and resets Q_1 and Q_2 by iterating over the whole current division.

Let us now describe how the algorithm maintains the invariants (Q1) and (Q2). Invariant (Q1) is handled easily by storing a flag in each row and column record whether it is currently queued for merging. Towards maintaining invariant (Q2), observe that a pair of adjacent rows R_1 and R_2 can suddenly become mergeable within the division $(\mathcal{R}'_i, \mathcal{C}'_i)$ in five different ways (the situation for columns is analogous):

(T1) one of R_1, R_2 was obtained by merging two rows in the previous step,

(T2) at least one of R_1, R_2 contains a non-empty cell in a column that was obtained by merging two columns in the previous step

(T3) at least one of R_1, R_2 was tall with respect to $(\mathcal{R}'_{i+1}, \mathcal{C}'_{i+1})$ and is no longer tall with respect to $(\mathcal{R}'_i, \mathcal{C}'_i)$, and

(T4) at least one of R_1, R_2 was dense in the previous step but $i = m_j$ for some $j \in [\ell]$ and the algorithm enters next phase.

(T5) the parameter d was doubled at the beginning of the current step.

The easiest cases to handle are (T4) and (T5) because the algorithm recomputes the queues Q_1 and Q_2 from scratch whenever it doubles d or enters the next phase. In the cases (T1) and (T2), one of R_1, R_2 must contain a non-empty cell that was involved in the last merge. This will be detected and handled locally when merging rows or columns. The only case that requires extra care to handle globally is case (T3).

A tall row R can appear only by merging two non-tall rows. Observe that R ceases to be tall in a future division $(\mathcal{R}'_i, \mathcal{C}'_i)$ where i is the largest integer for which $|R| \leq \frac{\delta n}{i}$ holds. Therefore, we can immediately compute this index i upon the creation of R as $i = \lfloor \frac{\delta n}{|R|} \rfloor$. The algorithm stores an extra array B of length n where $B[i]$ is a list that collects precisely all rows (resp. columns) that are tall (resp. wide) within $(\mathcal{R}'_{i+1}, \mathcal{C}'_{i+1})$ but no longer within $(\mathcal{R}'_i, \mathcal{C}'_i)$. So whenever a tall row R (resp. wide column) is obtained by merging, it is added to the list $B[i]$ for $i = \lfloor \frac{\delta n}{|R|} \rfloor$. Then in the j-th step, the algorithm first iterates over the list $B[j]$, checking whether some of the rows (resp. columns) therein can be merged with their neighbors and adding all such pairs to the queue Q_1

(resp. Q_2). Observe that this does not incur any additional overhead to the overall runtime since we detect each tall row (resp. wide column) and store it in $\mathcal{O}(1)$ time upon its creation and it is then once removed from the array B, again at constant cost.

This concludes the global overview of the implementation. Now, we proceed to specify the data stored for each row, column and non-empty cell which then allow an implementation of MergeRows and MergeCols methods in linear time with respect to the non-empty cells involved in the operation.

Row/column structure. The structure for columns and rows is analogous and we describe its semantics only for rows. Formally, each row R is represented by a record Strip consisting of the following:

- prev and next: Pointers to the previous and next row in the top-to-bottom order (or `null` if this is the first or the last row). Thus, all rows form a doubly-linked list.
- id: Unique identifier of the row from the set $[n]$ that respects the top-to-bottom order of rows, i.e., we have $R.\mathsf{id} < R.\mathsf{next}.\mathsf{id}$ whenever $R.\mathsf{next} \neq \texttt{null}$.
- cells: Number of non-empty cells in R.
- firstCell: Pointer to the record of the leftmost non-empty cell in the row.
- size: The height of the row $|R|$, i.e., the number of original rows from M contained in R.
- inQueue: Boolean flag whether this row is already queued for merging. That is $R.\mathsf{inQueue} =$ `true` if R appears in Q_1 and `false` otherwise.
- density: Number of rows of the previous division $(\mathcal{R}'_{m_j}, \mathcal{C}'_{m_j})$ contained in R.
- cellsWithNext: Number of non-empty cells in the union of R with $R.\mathsf{next}$ or ∞ if R is the last row ($R.\mathsf{next} = \texttt{null}$).

Cell structure. Each non-empty cell C is represented by a record Cell consisting of the following:

- left and right: Pointers to the previous and next non-empty cell in the same row, or `null` if there is no such cell.
- below and above: Pointers to the previous and next non-empty cell in the same column, or `null` if there is no such cell.
- row and column: Pointers to the row and column containing the cell C.

Merging operation. We describe the implementation of $\mathsf{MergeRows}(R_1, R_2)$ in linear time with respect to the number of non-empty cells in R_1 and R_2, the merging of columns is analogous. First, we initialize a record Strip for a new row R and replace R_1 and R_2 with R by setting the fields prev and next appropriately at R and its at most two adjacent rows. Second, we set $R.\mathsf{id}$ to $R_1.\mathsf{id}$, $R.\mathsf{size}$ to $R_1.\mathsf{size} + R_2.\mathsf{size}$ and $R.\mathsf{density}$ to $R_1.\mathsf{density} + R_2.\mathsf{density}$.

Next, we simultaneously traverse in left-to-right order the non-empty cells in both rows R_1 and R_2 that can be accessed through $R_1.\mathsf{firstCell}$ and $R_2.\mathsf{firstCell}$. Let us first describe the traversal ignoring the necessary updates to cellsWithNext fields. Crucially, for a pair of cells C_1 in R_1 and C_2 in R_2 we can determine in constant time whether they share the same column or which one is more to the left by comparing $C_1.\mathsf{column}.\mathsf{id}$ with $C_2.\mathsf{column}.\mathsf{id}$. This allows the procedure to simultaneously traverse cells in both rows in the left-to-right order while relinking pointers or replacing two non-empty cells in the same column with a single new non-empty cell. Along the way, it counts the number of non-empty cells in R and sets $R.\mathsf{cells}$ appropriately afterwards.

However, we still need to handle updating cellsWithNext fields in rows R, R.prev (if it exists) and potentially every column such that the cell in its intersection with R is non-empty (cf. (T1) and (T2)). To arrange the former, the procedure traverses R at most two more times at the end, once per each adjacent row to update cellsWithNext fields and potentially add R with either of its neighbors to Q_1 for merging. For the latter, we modify the merging traversal of R_1 and R_2 to always consider 2 non-empty cells from both R_1 and R_2. Out of these (at most) four cells, it is decidable in constant time how cellsWithNext changes for the affected columns and in extension, whether they should be added to Q_2 for merging.

Running time. Finally, let us bound the running time. First, the algorithm must construct the initial division $(\mathcal{R}'_n, \mathcal{C}'_n)$ out of the π-avoiding permutation σ on input. This can be done straightforwardly in linear time using access to σ in sorted orders by both indices and values. This takes $\mathcal{O}(n \lg c_\pi)$ time using the optimal algorithm for sorting pattern-avoiding sequences [Opl24].

In the following, let $d_i = 2^i$ denote the value of d after doubling i times and let $i_{\max}$ be the index such that the algorithm finished with the value of d equal to $d_{i_{\max}}$. Recall that we have $i_{\max} \in \mathcal{O}(\lg c_\pi)$ since $d_{i_{\max}} \leq 10c_\pi$. We denote by $n_1, \ldots, n_{i_{\max}}$ the sizes of divisions where the algorithm doubled the parameter d.

We claim that $n_i \in \mathcal{O}(n/d_i)$. To show this, focus at the step when the doubling $d_{i-1} \to d_i$ occurred. First, observe that we have $n_i \geq 2$ since the algorithm have not finished at that point yet. By our previous arguments, there are at most n_i/δ tall rows and at most n_i/δ dense rows. This leaves at least $\lfloor \frac{n_i}{2} \rfloor - \frac{2}{\delta} n_i \geq \frac{3}{20} n_i$ pairs of adjacent rows that are neither tall nor dense. Since none of these pairs can be merged, each such pair contains together strictly more than d_{i-1} non-empty cells. Therefore, there are at least $d_{i-1} \cdot \frac{3}{20} n_i \geq \frac{1}{20} d_i n_i$ non-empty cells in total and since the number of non-empty cells is trivially at most n, we obtain $n_i \leq \frac{n}{20 d_i} \in \mathcal{O}(n/d_i)$.

Let us now bound all the time incurred during every doubling of the parameter d. Every doubling of d causes an iteration over all rows and columns of the current division and adding mergeable pairs to the respective queues. Therefore, this takes time linear in the number of rows and columns which makes $\mathcal{O}(\frac{n}{d_1} + \cdots + \frac{n}{d_{i_{\max}}}) \subseteq \mathcal{O}(n)$ time in total.

Next, we bound the total time taken by all calls to MergeRow and MergeColumn. Recall that a single call takes linear time with respect to the number of non-empty cells participating in the merge. We fix $i \in [i_{\max}]$ and derive an upper bound on all such calls that occurred when d was equal to $d_i = 2^i$. After the doubling step $d_{i-1} \to d_i$, there were at most $\mathcal{O}(n/d_i)$ rows and columns. The algorithm only merges rows and columns if their union contains at most d_i non-empty cells which makes a single call to take $\mathcal{O}(d_i)$ time. Moreover, each merge decreases either the number of rows or columns by one and thus, there are at most $\mathcal{O}(n/d_i)$ merges with combined runtime of $\mathcal{O}(n)$. Together, this bounds the total time of all merging to $\mathcal{O}((i_{\max} + 1) \cdot n) = \mathcal{O}((\lg c_\pi + 1) \cdot n)$.

Finally, it remains to bound the time needed to copy and output the divisions $(\mathcal{R}_j, \mathcal{C}_j)$ for every $j \in [\ell]$ and all the other extra work done at the end of each phase. At the end of the j-th phase, the algorithm writes out the current division of size $\mathcal{O}(c_\pi \cdot (|\mathcal{R}_j| + |\mathcal{C}_j|)) = \mathcal{O}(c_\pi \cdot m_j)$ and iterates over all rows and columns in $\mathcal{O}(m_j)$ time. In total, this takes $\mathcal{O}(c_\pi \cdot \sum_{j=1}^{\ell} m_j)$ time as promised.

7 Proof of the treewidth structural lemma

In this section, we prove the balanced decomposition result for permutations of bounded treewidth (Lemma 5.1).

Grid-width. In order to go from the treewidth of incidence graph G_σ to divisions, we exploit a functionally equivalent parameter *grid-width* [JOV24]. For any subset S of 1-entries in M_σ, the *grid complexity* of S is the smallest number d such that there are at most d column intervals and at most

d row intervals such that S lies in the intersection of these rows and columns while no other 1-entry lies in these columns or rows. Informally, there exists a division $(\mathcal{R},\mathcal{C})$ where S is precisely a union of several components that together span at most d rows and d columns. Observe that if S and T are sets of grid complexity at most d, then their union $S \cup T$ and difference $S \setminus T$ both have grid complexity at most $2d$.

A *grid tree* T of σ is a rooted tree with n leaves, each leaf being marked with a distinct entry (i, σ_i) of σ. For a node v in T, let S_v^T denote the entries of τ that appear as leaf labels in the subtree of T rooted in v. The grid-width of T, denoted by $\mathrm{gw}^T(\sigma)$ is the maximum grid complexity of S_v^T over all nodes of T. And as usual, *grid-width* of σ, denoted by $\mathrm{gw}(\sigma)$, is the minimum $\mathrm{gw}^T(\sigma)$ over all grid trees of σ. Observe that we can without loss of generality assume that every inner node in T has exactly two children. It is known that for every permutation σ, we have $\frac{1}{8}\,\mathrm{tw}(G_\sigma) \leq \mathrm{gw}(\sigma) \leq \mathrm{tw}(G_\sigma) + 2$, where tw denotes treewidth. Moreover, there is a linear-time algorithm that receives a tree decomposition of width k for G_σ on input and outputs a grid tree T such that $\mathrm{gw}^T(\sigma) \leq k + 2$ [JOV24, Proposition 4.1].

As a warm-up, let us argue that a grid tree T for τ of small grid-width d can be easily turned into a hierarchy of divisions with small components if we ignore the balancedness constraints ((c) and (e)). We again start with the trivial division into single rows and columns of M_τ. Our strategy is to contract edges adjacent to leaves in T one by one until we arrive at a singleton tree, and mirror these contractions in the divisions. Along the way, each node v in the tree has its associated set of τ-entries $S(v)$ such that (i) the grid complexity of $S(v)$ is at most $3d$, and (ii) the sets $S(v)$ form a partition of τ. At the beginning, we set $S(v)$ to be empty for the internal nodes of T and for a leaf v, we set $S(v) = \{(i, \tau_i)\}$ where (i, τ_i) is the label of v. In every step, we choose a leaf v with parent w in the current tree, remove the leaf v and set $S(w) \leftarrow S(v) \cup S(w)$.

Let us argue that the grid complexity of the sets $S(v)$ remains bounded throughout the process. To see that, notice that if we trace the contractions back, each node v corresponds to a connected subtree T_v of the original grid tree T. Moreover, this subtree T_v can be obtained from T by taking a subtree rooted in some node v_1 and then removing at most two of its rooted subtrees since v has at most two children. But then $S(v)$ is equal to $S_{v_1}^T \setminus (S_{v_2}^T \cup S_{v_3}^T)$ where all $S_{v_1}^T, S_{v_2}^T, S_{v_3}^T$ are sets of grid complexity at most d. It follows that $S(v)$ has grid complexity at most $3d$.

Finally, we describe how we update the division at each contraction. Throughout, we maintain the smallest division such that for every two different nodes v and w, the sets $S(v)$ and $S(w)$ occupy pairwise different rows and columns. Therefore, when we delete a leaf v with a parent w, we merge every adjacent pair of rows and columns such that one of them intersects $S(v)$ and the other $S(w)$. This guarantees that every component eventually spans at most $3d$ rows and columns of the division because of our bound on the grid complexity of the sets $S(v)$. However, that does not directly bound the number of rows and columns occupied by a single component throughout the whole process. That is because after contracting in T, we start merging rows and columns one by one and thus, it might happen that the new component initially spans $2 \cdot 3d$ rows and columns before being pushed down to at most $3d$ through row and column merges.

We omit detailed description of an efficient implementation and include it only for the full, balanced, variant.

From a very high-level view, we first compute a grid tree T of τ with small grid-width that we then use to guide the creation of hierarchy of divisions. In the general decomposition lemma, we performed merging operations on adjacent rows and columns greedily as long as they maintained given invariants. Here, we use the grid tree T as a guide to which pairs of columns and rows can be merged at each step similarly to the warm-up. However, we allow contraction of not only the edges incident to leaves but also edges incident to vertices with a single child. This procedure follows the common approach for computing a balanced treewidth decomposition [BH98] based on parallel tree contractions [MR89]. In this way, there is still enough freedom in choosing the next contraction

which allows us to maintain the balancedness constraints in the same vein as before.

The first step of the decomposition is to get a grid tree of low grid-width. To do that, we invoke any single exponential approximation algorithm for treewidth – say the 2-approximation due to Korhonen [Kor21] – and immediately turn it into a grid tree with grid-width t' where $t' \leq 2\,\mathrm{tw}(G_\sigma) + 2$ [JOV24, Proposition 4.1].

Instead of computing all the divisions $(\mathcal{R}_1, \mathcal{C}_1), \ldots, (\mathcal{R}_\ell, \mathcal{C}_\ell)$ individually, the algorithm again just dynamically iterates through a sequence of divisions $(\mathcal{R}'_n, \mathcal{C}'_n), \ldots, (\mathcal{R}'_1, \mathcal{C}'_1)$ such that

(i) The division $(\mathcal{R}'_i, \mathcal{C}'_i)$ is a coarsening of $(\mathcal{R}'_{i+1}, \mathcal{C}'_{i+1})$.

(ii) The division $(\mathcal{R}'_i, \mathcal{C}'_i)$ has exactly i rows and columns.

(iii) Each row (resp. column) of $M(\mathcal{R}'_i, \mathcal{C}'_i)$ has height (resp. width) at most $96t' \cdot n/i$.

(iv) Each row and column in $M(\mathcal{R}'_i, \mathcal{C}'_i)$ is obtained from merging at most $192t' \cdot m_{j+1}/m_j$ rows or columns of $M(\mathcal{R}'_{m_{j+1}}, \mathcal{C}'_{m_{j+1}})$, where j is the largest integer such that $m_j \leq i$.

(v) Each component in $(\mathcal{R}'_i, \mathcal{C}'_i)$ spans at most $12t + 12$ rows and columns.

We get the desired output by copying out $(\mathcal{R}_i, \mathcal{C}_i) = (\mathcal{R}'_{m_i}, \mathcal{C}'_{m_i})$ for every $i \in [\ell]$.

We start with $(\mathcal{R}'_n, \mathcal{C}'_n)$ being the trivial (finest) division where each row (resp. column) contains exactly one row (resp. column) of the input matrix M. As in the warm-up, we associate to each node v in the tree T a set $S(v)$, initially set to consist of the labels in leaves and empty elsewhere. The division at hand is maintained as a smallest division with equal number of rows and columns that separates the sets $S(\cdot)$, i.e., there is no column or row that intersect $S(v), S(w)$ for two different vertices. The run is again split into $\ell + 1$ phases with the j-th phase generating divisions $(\mathcal{R}'_i, \mathcal{C}'_i)$ for $i \in \{m_j, \ldots, m_{j+1} - 1\}$. However, the size and density conditions are now imposed on the sets $S(\cdot)$ instead of rows and columns of the division. A vertex v in T is *large* if $|S(v)| > 8 \cdot n/|V(T)|$ where $V(T)$ is the set of vertices in T. Moreover, v is *dense* in the j-th phase if $S(v)$ is a union of more than $192t' \cdot m_{j+1}/m_j$ rows or columns of the division $(\mathcal{R}'_{m_{j+1}}, \mathcal{C}'_{m_{j+1}})$.

We describe row and column merges in batches which are performed in immediate succession. However, if a division $(\mathcal{R}'_i, \mathcal{C}'_i)$ of size exactly $i = m_j$ for some $j \in [\ell]$ is encountered, the algorithm outputs the current division and resets density of vertices in T before continuing with individual row and column merges. In one step, the algorithm finds a vertex v with parent w such that v is either a leaf or has a single child and, moreover, v and w are both neither large nor dense. We contract the edge $\{v, w\}$ in T and set $S(v') \leftarrow S(v) \cup S(w)$ where v' is the vertex obtained in the contraction. Afterwards, the algorithm keeps merging adjacent rows and columns that contain $S(v')$ where crucially, it alternates between merging rows and columns and stops when another row or column merge is no longer possible. The reason for this is to maintain the equality between number of rows and columns in the divisions.

This process continues all the way until a trivial division $(\mathcal{R}'_1, \mathcal{C}'_1)$ is reached.

Correctness. We have to verify that the algorithm is guaranteed to finish, i.e., there will always be a suitable pair of vertices v, w in T to merge, and properties (i)–(v) hold for every division $(\mathcal{R}'_i, \mathcal{C}'_i)$.

First, we show that (i)–(v) hold for every division. Properties (i) and (ii) are maintained directly by the algorithm. Towards showing (v), recall that for a vertex v in the tree T we denote by T_v the connected subtree of the original grid tree that was contracted into v. We maintain that every vertex in T has at most two children and thus, T_v is obtained by taking a rooted subtree and then removing up to two of its rooted subtrees. Therefore, $S(v)$ is equal to the set $S^T_{v_1}$, $S^T_{v_1} \setminus S^T_{v_2}$, or $S^T_{v_1} \setminus (S^T_{v_2} \cup S^T_{v_3})$ for some v_1, v_2, v_3 in the original grid tree and the grid complexity of $S(v)$ is at most $3t' \leq 6t + 6$. As in the warm-up, this implies that in the worst case a single component can

contain at most $2 \cdot 3t' = 12t + 12$ rows and columns following some edge contraction before being pushed down to at most $3t' = 6t + 6$ through row and column merges.

Properties (iii) and (iv) follow similarly to their analogues in Lemma 2.2. Specifically, we have $|S(v)| \leq 16n/|V(T)|$ for every vertex v in T since we never contract vertices with $S(v) > 8n/|V(T)|$ and $|V(T)|$ only decreases throughout the process. We obtain $|S(v)| \leq 96t'n/i$ by plugging in the inequality $|V(T)| \geq i/(6t')$ that is due to (v). For analogous reasons, any component in the j-th phase occupies at most $192 \cdot m_{j+1}/m_j$ rows and columns of the division $(\mathcal{R}'_{m_{j+1}}, \mathcal{C}'_{m_{j+1}})$ and the same bound naturally transfers to every individual row or column in the component.

Finally, we show that the tree T always contains a suitable pair of vertices to merge. Assume for a contradiction that there was no suitable pair of vertices in T to merge at some step and let n' denote the number of vertices in T at that moment. As we already argued, there are at least $n'/2$ vertices with at most one child. Each such vertex is a candidate for merging together with its parent. Moreover, each vertex participates in at most two such mergeable pairs of adjacent vertices – this happens when both children of a vertex are leaves, or there is vertex that has a single child which is itself a degree-2 vertex or a leaf. Thus, there are at least $n'/4$ disjoint pairs of vertices v, w in T such that w is a parent of v and v is either a leaf or has a single child.

Let us denote by L and D the number of large vertices and dense vertices, respectively. We have $L < n'/8$ since $|S(v)| > 8n/n'$ for every large vertex v. The number of dense vertices can be bounded as

$$D < \frac{2m_{j+1}}{96t' \frac{m_{j+1}}{m_j}} \leq \frac{m_{j+1}}{48t' \frac{m_{j+1}}{m_j}} \leq \frac{m_j}{48t'} \leq \frac{n'}{8}$$

where the first inequality holds since each dense component contains strictly more than $96t' \frac{m_{j+1}}{m_j}$ rows or columns of $(\mathcal{R}'_{m_{j+1}}, \mathcal{C}'_{m_{j+1}})$ and the third inequality holds since the number of rows and columns in the current division is at least m_j and at most $6t'n'$ (and, thus, $m_j \leq 6t'n'$). As a result, there are strictly less than $n'/4$ large and dense vertices in T so there is always at least one possible edge that can be contracted.

Implementation. The algorithm can be implemented similarly to the general decomposition lemma with the advantage that we can afford more dependence on the treewidth t in the runtime. This is because the initial approximation of treewidth already takes $2^{\mathcal{O}(t)}n$ time and thus, we only aim at a runtime $\mathcal{O}_t(n)$. Therefore, we do not describe the implementation in full detail and mostly refer to the implementation in Lemma 2.2.

Recall that the algorithm in Lemma 2.2 maintains two queues Q_1 and Q_2 that store adjacent pairs of rows and columns for future merging. Since we contract edges in the tree T, we now maintain a single queue of edges Q available for contraction together with size, density information stored in each vertex of T. This allows us to spend only $\mathcal{O}(1)$ time to pop the next edge for contraction, update T and potentially add the new vertex with some of its (at most three) neighbors to Q. As before, we maintain a bucket structure to keep track of large vertices and reintroduce them back when they cease to be large, without additional overhead. For the density, we allow a complete iteration over the tree at the end of each phase which resets the density of each dense vertex.

We store the division in the same data structure, only omitting size, inQueue, density and cellsWithNext fields in every row and column. Additionally, each vertex v in T stores pointers to every row and column that is intersected by $S(v)$. After contracting an edge $\{v, w\}$ in T, the algorithm iterates through the rows and columns containing $S(v), S(w)$ and merges them one by one alternating between rows and columns. This can be straightforwardly implemented in $\mathcal{O}(t^2)$ time where one factor comes from the number of rows and columns occupied by each component and the other from merging individual adjacent pairs of rows and columns. Overall, there are $\mathcal{O}(n)$ edge contractions each incurring $\mathcal{O}_t(1)$ time for an overall runtime of $\mathcal{O}_t(n)$.

References

[ABP06] Eyal Ackerman, Gill Barequet, and Ron Y Pinter. A bijection between permutations and floorplans, and its applications. *Discrete Applied Mathematics*, 154(12):1674–1684, 2006.

[AR08] Shlomo Ahal and Yuri Rabinovich. On complexity of the subpattern problem. *SIAM J. Discret. Math.*, 22(2):629–649, 2008. doi:10.1137/S0895480104444776.

[Arr99] Richard Arratia. On the stanley-wilf conjecture for the number of permutations avoiding a given pattern. *Electron. J. Comb.*, 6, 1999. doi:10.37236/1477.

[Bar13] Jérémy Barbay. From time to space: Fast algorithms that yield small and fast data structures. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2013. doi:10.1007/978-3-642-40273-9_8.

[BBGT24] Edouard Bonnet, Romain Bourneuf, Colin Geniet, and Stéphan Thomassé. Factoring pattern-free permutations into separable ones. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 752–779, 2024. doi:10.1137/1.9781611977912.30.

[BBL98] Prosenjit Bose, Jonathan F. Buss, and Anna Lubiw. Pattern matching for permutations. *Inf. Process. Lett.*, 65(5):277–283, 1998. doi:10.1016/S0020-0190(97)00209-3.

[BEMS24] Omri Ben-Eliezer, Slobodan Mitrović, and Pranjal Srivastava. Approximate counting of permutation patterns. *arXiv preprint arXiv:2411.04718*, 2024.

[BFCP+05] Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[BH98] Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM J. Comput.*, 27(6):1725–1746, 1998. doi:10.1137/S0097539795289859.

[BKM21] Benjamin Aram Berendsohn, László Kozma, and Dániel Marx. Finding and counting permutations via csps. *Algorithmica*, 83(8):2552–2577, 2021. doi:10.1007/s00453-021-00812-z.

[BKO24] Benjamin Aram Berendsohn, László Kozma, and Michal Opler. Optimization with pattern-avoiding input. In Bojan Mohar, Igor Shinkar, and Ryan O'Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 671–682. ACM, 2024. doi:10.1145/3618260.3649631.

[BKRT22] Édouard Bonnet, Eun Jung Kim, Amadeus Reinald, and Stéphan Thomassé. Twin-width VI: the lens of contraction sequences. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 1036–1056. SIAM, 2022. doi:10.1137/1.9781611977073.45.

[BKTW21] Édouard Bonnet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width I: tractable FO model checking. *ACM Journal of the ACM (JACM)*, 69(1):1–46, 2021. doi:10.1145/3486655.

[BN13] Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theor. Comput. Sci.*, 513:109–123, 2013. URL: https://doi.org/10.1016/j.tcs.2013.10.019, doi:10.1016/J.TCS.2013.10.019.

[BNdM+21] Édouard Bonnet, Jaroslav Nešetřil, Patrice Ossona de Mendez, Sebastian Siebertz, and Stéphan Thomassé. Twin-width and permutations, 2021. arXiv:2102.06880.

[Bón22] Miklós Bóna. *Combinatorics of permutations*. CRC Press, 2022.

[BV94] Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994. doi:10.1016/S0022-0000(05)80002-9.

[CGK+15] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Pattern-avoiding access in binary search trees. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015*, pages 410–423. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.32.

[Cha88] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988. doi:10.1137/0217026.

[Cib09] Josef Cibulka. On constants in the Füredi–Hajnal and the Stanley–Wilf conjecture. *Journal of Combinatorial Theory, Series A*, 116(2):290–302, 2009. doi:10.1016/j.jcta.2008.06.003.

[CJKS24] Sankardeep Chakraborty, Seungbum Jo, Geunho Kim, and Kunihiko Sadakane. Succinct data structures for baxter permutation and related families. In Julián Mestre and Anthony Wirth, editors, *35th International Symposium on Algorithms and Computation, ISAAC 2024, December 8-11, 2024, Sydney, Australia*, volume 322 of *LIPIcs*, pages 17:1–17:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. URL: https://doi.org/10.4230/LIPIcs.ISAAC.2024.17, doi:10.4230/LIPICS.ISAAC.2024.17.

[CLP11] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In Ferran Hurtado and Marc J. van Kreveld, editors, *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, pages 1–10. ACM, 2011. doi:10.1145/1998196.1998198.

[CPY24] Parinya Chalermsook, Seth Pettie, and Sorrachai Yingchareonthawornchai. Sorting pattern-avoiding permutations via 0-1 matrices forbidding product patterns. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 133–149. SIAM, 2024. doi:10.1137/1.9781611977912.7.

[EZL21] Chaim Even-Zohar and Calvin Leng. Counting small permutation patterns. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2288–2302. SIAM, 2021.

[FFNO11] Stefan Felsner, Éric Fusy, Marc Noy, and David Orden. Bijections for baxter families and related objects. *Journal of Combinatorial Theory, Series A*, 118(3):993–1020, 2011.

[FH92] Zoltán Füredi and Péter Hajnal. Davenport-Schinzel theory of matrices. *Discret. Math.*, 103(3):233–251, 1992. doi:10.1016/0012-365X(92)90316-8.

[FH11] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

[Fox13] Jacob Fox. Stanley-wilf limits are typically exponential, 2013. arXiv:1310.8378.

[Fre76] Michael L. Fredman. How good is the information theory bound in sorting? *Theor. Comput. Sci.*, 1(4):355–361, 1976. doi:10.1016/0304-3975(76)90078-5.

[GM14] Sylvain Guillemot and Dániel Marx. Finding small patterns in permutations in linear time. In *ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*, pages 82–101. SIAM, 2014. doi:10.1137/1.9781611973402.7.

[Gol09] Alexander Golynski. Cell probe lower bounds for succinct data structures. In Claire Mathieu, editor, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 625–634. SIAM, 2009. doi:10.1137/1.9781611973068.69.

[GR22] Pawel Gawrychowski and Mateusz Rzepecki. Faster exponential algorithm for permutation pattern matching. In *5th Symposium on Simplicity in Algorithms, SOSA@SODA 2022, Virtual Conference, January 10-11, 2022*, pages 279–284. SIAM, 2022. doi:10.1137/1.9781611977066.21.

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.

[JMS04] Joseph F. JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In Rudolf Fleischer and Gerhard Trippen, editors, *Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20-22, 2004, Proceedings*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568. Springer, 2004. doi:10.1007/978-3-540-30551-4_49.

[JOV18] Vít Jelínek, Michal Opler, and Pavel Valtr. Generalized coloring of permutations. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, volume 112 of *LIPIcs*, pages 50:1–50:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. URL: https://doi.org/10.4230/LIPIcs.ESA.2018.50, doi:10.4230/LIPICS.ESA.2018.50.

[JOV24] Vít Jelínek, Michal Opler, and Pavel Valtr. Generalized coloring of permutations. *Algorithmica*, 86(7):2174–2210, 2024. URL: https://doi.org/10.1007/s00453-024-01220-9, doi:10.1007/S00453-024-01220-9.

[Kit11] Sergey Kitaev. *Patterns in permutations and words*, volume 1. Springer, 2011.

[Kla00] Martin Klazar. The Füredi-Hajnal Conjecture Implies the Stanley-Wilf Conjecture. In *Formal Power Series and Algebraic Combinatorics: 12th International Conference, FPSAC'00*, pages 250–255. Springer, 2000. doi:10.1007/978-3-662-04166-6_22.

[Knu68] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[Kor21] Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 184–192. IEEE, 2021. doi:10.1109/FOCS52979.2021.00026.

[MR89] Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. *Adv. Comput. Res.*, 5:47–72, 1989.

[MRRR12] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theor. Comput. Sci.*, 438:74–88, 2012. URL: https://doi.org/10.1016/j.tcs.2012.03.005, doi:10.1016/J.TCS.2012.03.005.

[MT04] Adam Marcus and Gábor Tardos. Excluded permutation matrices and the Stanley–Wilf conjecture. *Journal of Combinatorial Theory, Series A*, 107(1):153–160, 2004. doi:10.1016/J.JCTA.2004.04.002.

[Nav16] Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016. URL: http://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/compact-data-structures-practical-approach?format=HB.

[Opl24] Michal Opler. An optimal algorithm for sorting pattern-avoiding sequences. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, pages 689–699. IEEE, 2024. doi:10.1109/FOCS61266.2024.00049.

[Păt07] Mihai Pătraşcu. Lower bounds for 2-dimensional range counting. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 40–46. ACM, 2007. doi:10.1145/1250790.1250797.

[PSZ22] Michal Pilipczuk, Marek Sokołowski, and Anna Zych-Pawlewicz. Compact representation for matrices of bounded twin-width. In Petra Berenbrink and Benjamin Monmege, editors, *39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022, March 15-18, 2022, Marseille, France (Virtual Conference)*, volume 219 of *LIPIcs*, pages 52:1–52:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.STACS.2022.52, doi:10.4230/LIPICS.STACS.2022.52.

[TOG17] Csaba D. Tóth, Joseph O'Rourke, and Jacob E. Goodman. *Handbook of discrete and computational geometry*. CRC press, 2017.

[Wes95] Julian West. Generating trees and the catalan and schröder numbers. *Discret. Math.*, 146(1-3):247–262, 1995. doi:10.1016/0012-365X(94)00067-1.