Compact Representations of Pattern-Avoiding Permutations

László Kozma
Faculty of Computer Science, TU Dresden, Germany

Michal Opler
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 bounds that hold 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) for 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) obtained for separable permutations via different techniques. All our data structures can be constructed in linear time.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Mathematics of computing → Combinatorics

Keywords and phrases pattern-avoiding permutations, compact data structures, succinct data structures, range counting, twin-width

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.92

Related Version *Full Version*: `https://arxiv.org/abs/2510.20382`

Funding *Michal Opler*: 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).

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 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 [32, 10]). 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 [34, § 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 $i \in [n]$, using $n\lceil \lg n \rceil$ bits, essentially matching the information-theoretic lower bound. However, to turn 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 [36], supporting $\tau(i)$ and $\tau^{-1}(i)$, and in fact, arbitrary power-k queries $\tau^k(i)$, in time $\mathcal{O}(1/\varepsilon)$. Golynski [28] 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 [37] for a broad survey).

Data structures with a stricter space requirement, having only a lower order additive overhead, are called *succinct*. Munro et al. [36] also design succinct schemes for representing permutations, i.e., using $n \lg n + o(n \lg n)$ bits, but with query costs necessarily super-constant.

All the mentioned solutions are, however, too wasteful for storing pattern-avoiding permutations. As the landmark result of Marcus and Tardos [35] 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 $s_\pi = \lim_{n\to\infty} \sqrt[n]{|\mathrm{Av}_n(\pi)|}$ [2].[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 [25] 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 [38]. 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 [16] 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

[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 is used when referring to queries.

[2] The result, by Marcus and Tardos [35], also builds on work by Füredi and Hajnal [26], and Klazar [33].

[3] The growth rate of s_π as a function of $k = |\pi|$ is far from fully understood, it is known [24] that $s_\pi \in 2^{O(k)}$ for all π, and $s_\pi \in 2^{\Omega(k^{1/4})}$ for some π.

and 3142 [15]. These permutations also have a natural binary tree representation (they can be recursively constructed via the *sum* and *skew sum* operators [15, 16, 8]) and Chakraborty et al. exploit this structure to build their efficient representation.

In a similar way, Chakraborty et al. also design a data structure for storing *Baxter-permutations*, although with polylogarithmic query times. Baxter permutations are not defined using classical pattern-avoidance, but via the concept of *vincular patterns*. They also admit a certain tree-representation (by min-/max- Cartesian trees), which is used in [16]. 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.** *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 related technique was introduced in [8], 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, and we believe that given its strong balancedness properties, it may have further applications.

Our data structure also supports, in time $\mathcal{O}(1)$ and with negligible (sublinear) 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, we use 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 also make the optimal $\lg s_\pi$ factor explicit. We note that for the space bound the $\mathcal{O}$-notation only hides a factor of two in the leading term, and an additive sublinear (in n) term of the form $\mathcal{O}_\pi(n/\lg^{1/4} n)$. Importantly, the avoided pattern π is not known during construction and operation, and only τ is given as input.

Halving the constant factor in the space bound, and thus making the data structure succinct, is a challenging open question. We 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 [1, 7]).

▶ **Theorem 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 [1], so Theorem 2 applies. 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 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 [43], so their succinct data structure uses $n \cdot \lg(3 + 2\sqrt{2}) + o(n) \approx 2.54n$ bits. As in the case of Theorem 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 [16], where some of the operations needed $\Theta(\lg\lg n)$ time.

We note that the query times in Theorem 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 [35] 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 is an early example. Fine-grained complexity bounds for pattern-avoiding inputs were obtained for several algorithmic problems. These include: searching in binary trees with rotation [17, 18, 8], the k-server and the traveling salesman problems [8], factorization [11], permutation pattern matching and counting [15, 1, 29, 7, 27, 22], approximate counting [5, 39], and sorting [38].

Our data structure for storing permutations has further algorithmic applications. In particular, it can be extended to support *geometric queries* such as (rectangle) *range minimum*, or (rectangle) *range counting*, with the definition of patterns naturally extended from permutations to point sets (under a general position assumption). Our results show that pattern-avoidance in the input allows for speed-ups in these natural and well-studied problems.

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 [30, 9, 6, 23]. 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 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.*

In computational geometry, orthogonal range searching and its variants are among the best studied problems [42, § 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 [20, 31]. Pătraşcu [40] 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 [19]; 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 [4, 3]. 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 [3]. Thus, the recent linear-time sorting algorithm for pattern-avoiding permutations [38] naturally suggested the question of whether a compact data structure exists for this broad family of inputs, affirmatively answered by Theorem 1.

Bounded twin-width permutations and matrices. Twin-width is a recently emerging complexity measure for permutations, graphs, as well as more general structures [29, 13, 14]. 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 $2^{\mathcal{O}(|\pi|)}$ [29, 24]. We could therefore directly use a recent compact data structure for storing low twin-width matrices, by Pilipczuk, Sokołowski, and Zych-Pawlewicz [41]. 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 $\mathcal{O}(\lg \lg n)$, it supports only a subset of the queries we consider, and it takes 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 [41], and note some similarity in techniques, particularly in the proof of Theorem 3. In turn, our results also extend to storing permutations of *bounded twin-width* since every permutation of twin-width d avoids a certain pattern of length $(d+1)^2$ [29].

▶ **Corollary 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 recent result is that π-avoiding permutations can be expressed as a composition of $\mathcal{O}_\pi(1)$ many separables [11]. 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.[4] 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 5).

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 describe the details in § 3. 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.

Going from column- to row-coordinates, i.e., computing τ_i from i, poses multiple challenges. As a key step, we store the sub-permutations induced by each vertical strip of the 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-coordinate, 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 us 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.

Extensions. To support the geometric queries (Theorem 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 branching factors between consecutive levels. In this way, the amount of data stored at each level is kept small, with the running time bounded as $\mathcal{O}(\lg \lg n)$. We provide a brief sketch in § 3, see full version for the complete treatment.

For permutations of bounded treewidth (Theorem 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, which makes it possible to convert between row- and column-information. We refer to § 4 for more details with the complete argument in the full version.

Open questions. For our main result (Theorem 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.

[4] 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.

For our geometric data structure (Theorem 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.

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 $\mathcal{C}_n$ of permutations be obtained whenever sorting in $\mathcal{O}(\lg |\mathcal{C}_n|)$ time is possible?

2 Preliminaries

Permutation pattern-avoidance. A permutation $\tau = (\tau_1, \ldots, \tau_n)$ can be bijectively mapped 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 and column second, ordering top to bottom and left to right. We often 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 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 [35] showed that $\mathrm{ex}_\pi(n)$ is linear in n, for every fixed π. More precisely, there exists a quantity c_π so that 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 [26], 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 [21] 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 $[n]$. We refer to $(\mathcal{R}, \mathcal{C})$ as a *division* of M; intuitively, 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: for $\mathcal{R}, \mathcal{C}$ 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 call this 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 $(\mathcal{R}', \mathcal{C}')$.

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.

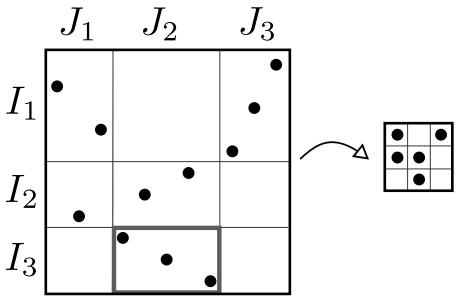

Figure 1 Permutation matrix M of size 11×11 (left) with division $(\mathcal{R}, \mathcal{C})$ and $M(\mathcal{R}, \mathcal{C})$ (right), with $\mathcal{R} = (I_1, I_2, I_3)$, $\mathcal{C} = (J_1, J_2, J_3)$, and cell $M[I_3, J_2]$ framed. Here, $I_1 = [1, 5], J_1 = [1, 3], I_2 = [6, 8], J_2 = [4, 8],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

- *sparse bitvector*: stores values $B[1], \ldots, B[n] \in \{0, 1\}$, supporting in $\mathcal{O}(1)$ time the operations:
  - $\mathsf{rank}(B, i)$: 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.

  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 [37, § 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, omitting the leaves in subtree of v.

  Standard implementations using $2n + o(n)$ bits can be found in [37, § 8].
- *function*: a mapping $A \to B$, with $A = [|A|]$ and $B = [|B|]$, with $|B|$ assumed to fit in a machine word. The space requirement of a standard implementation is $|A| \cdot \lceil \lg |B| \rceil$ bits.

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., [29]), significantly adapted and extended for our purposes.

▶ **Lemma 5** (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.

Due to space constraints, we defer the proof of Lemma 5 to the full version. A weaker statement of a similar flavor was obtained in [8]. In particular, Lemma 5 differs in enforcing an exact number of columns and rows in part (b), an additional balancedness condition between the layers in part (e) and a 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. 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 assume n is larger than some unspecified constant. Due to space constraints, some minor details are omitted here; refer to the full version for a complete treatment.

The data structure is based on two non-trivial levels of the balanced decomposition (Lemma 5). 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 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. Refer to Figure 2.

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. 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 from § 2 is thus $o(n)$.

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 us to identify the rank of the non-empty 2-cell containing the queried point, and relative rank information of the point within this 2-cell. More precisely, 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.

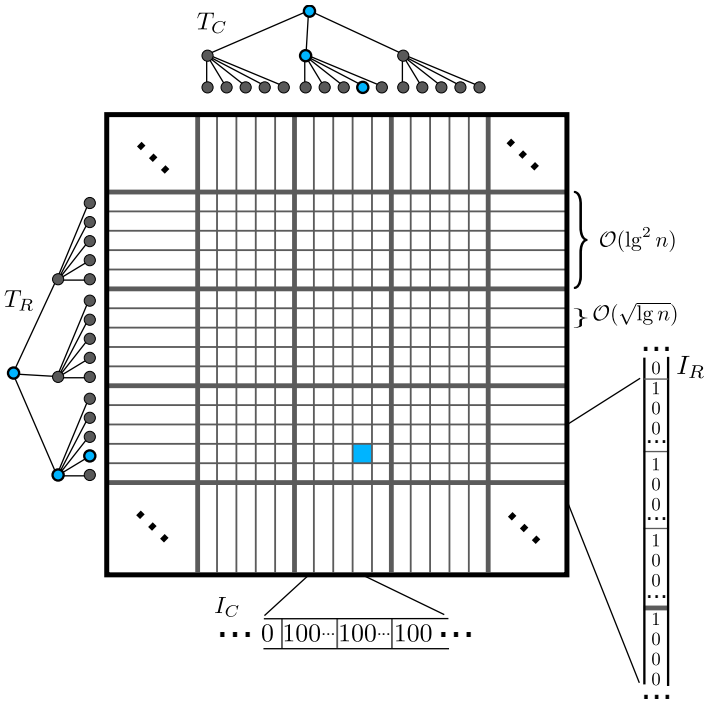

Figure 2 Decomposition of an $n \times n$ permutation matrix M with a (coarse) division $(\mathcal{R}_1, \mathcal{C}_1)$ that is a coarsening of a (fine) division $(\mathcal{R}_2, \mathcal{C}_2)$. Column and row trees T_C, T_R and column and row indices I_C, I_R illustrated. A 2-cell and corresponding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

The cell index c refers to the c-th non-zero cell in the column, its domain is thus $[10c_\pi]$, by Lemma 5(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 5(c). Storing $\mathsf{colPerm}$ 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 $\mathsf{colPerm}$ 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). This adds up, over all columns, to $n \lg s_\pi + o(n)$, accounted for in part D. As a refinement, for any subclass $\mathcal{C} \subseteq \mathrm{Av}(\pi)$, each column of width w_C can be encoded in $\lg |\mathcal{C}_{w_C}| + \mathcal{O}_\pi(\lg \lg n)$ bits; summing over all columns yields $\lg |\mathcal{C}_n| + o(n)$ bits, provided $\mathcal{C}$ is supermultiplicative.

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 $\mathcal{O}(s_\pi^w w^{10c_\pi} \cdot (w \lg(c_\pi w) + c_\pi w \lg w)) \subseteq o(n)$.

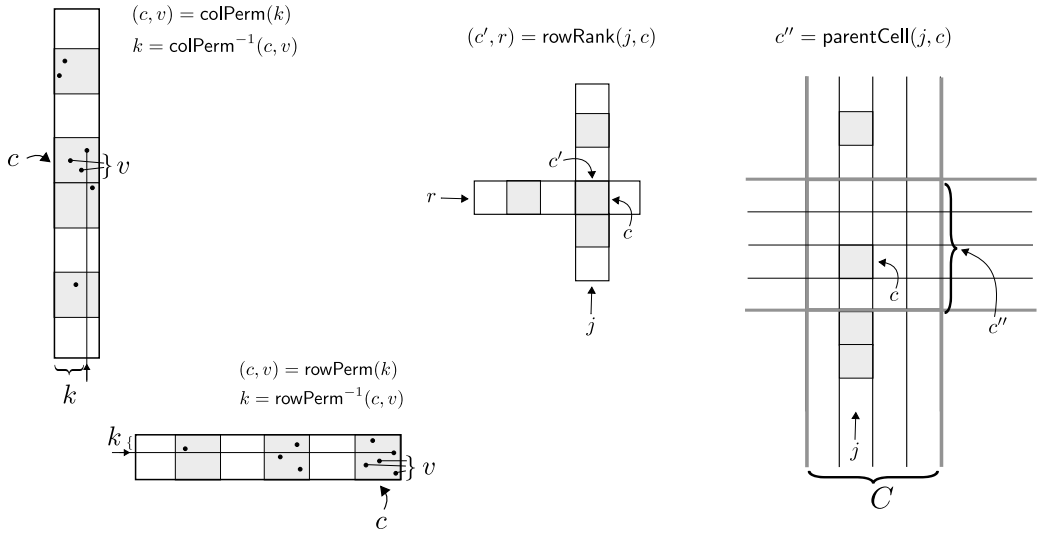

Figure 3 *Left*: column and row permutation mapping and their inverse. Value k is the absolute offset of the query within the column or row; c refers to the c-th non-zero cell within column or row (non-zero cells are shaded gray); v is the vertical rank of the query within its cell, i.e., v entries in 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.

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 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 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 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 colPerm that returns a pointer to a column permutation structure (as described in part C).

The implementation is recursive. Each node stores its precomputed query answers followed by variable-length pointers to its children's memory regions. The crucial observation is that the total size of a level-1 column's children (all $\mathcal{O}(\lg^{3/2} n)$ level-2 subcolumns) is $\mathcal{O}(\lg s_\pi \cdot \lg^2 n)$ bits, so each child pointer fits in $\mathcal{O}_\pi(\lg \lg n)$ bits. The precomputed rowRank and parentCell answers and the pointers each take $\mathcal{O}_\pi(\lg^{7/4} n)$ bits per level-1 node, giving $\lg s_\pi \cdot w + \mathcal{O}_\pi(\lg^{7/4} n)$ total per node. Summing over all nodes, the entire structure takes $\lg s_\pi \cdot n + \mathcal{O}_\pi(n/\lg^{1/4} n)$ bits, making this the space bottleneck of the data structure.

E. Recursive row structure G_R and row permutations rowPerm. This component is fully symmetric to parts C and D with rows and columns swapped. The interface provides subRow(j) navigating to the j-th subrow, colRank(j, c) returning the column identity and rank of the c-th non-zero cell of the j-th subrow, parentCell(j, c) returning the containing coarser cell, and rowPerm/rowPerm^{-1} mapping (for $\mathcal{R}_2$-rows) between row offsets and (cell index, within-cell rank) pairs. The implementation and space analysis are identical, giving $n \lg s_\pi + o(n)$ bits in total. Storing both the column and row structures accounts for the factor of two in the leading space term.

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 thus give textual description only for the former, with pseudocode in Figure 4.

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 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 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 s_2-th within the coarser column of $\mathcal{C}_1$ that contains it. The coarser column is 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 $\mathcal{C}_2$-column containing the query point. The $\mathcal{C}_2$-column is stored as a column permutation and we can identify, using the column offset s_3, the cell index c_2 and within-cell vertical rank v. This means that entry (τ_i, i) is in the c_2-th non-zero 2-cell in its $\mathcal{C}_2$-column and 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 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).

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 $r_i + r_3$

Figure 4 Implementation of the $i \to \tau(i)$ query.

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. Range minimum, next smaller, and related queries can all be supported with $o(n)$ additional space and $\mathcal{O}(1)$ time. The key idea is the same in each case: precompute query answers within individual fine columns/rows and within coarse cells (whose small size keeps the tables sublinear), then combine at most a constant number of precomputed answers to resolve the global query. See the full version for details.

Building the data structure. We briefly argue that the time to construct the data structure of Theorem 1 is linear. The main task is that of constructing the decomposition; by Lemma 5, 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 [38], 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.

Extension to geometric queries. Extension to complex geometric queries, e.g., *rectangle range counting* is more challenging. For this task, we build a hierarchy of $\mathcal{O}(\lg \lg n)$ divisions with block sizes decreasing at a polynomial rate, alongside the two-level hierarchy of Theorem 1. Queries decompose along the hierarchy, spending $\mathcal{O}(1)$ time per level for a total of $\mathcal{O}(\lg \lg n)$. The finest level of the new hierarchy coincides with $(\mathcal{R}_2, \mathcal{C}_2)$; at each coarser level the branching factor is $\mathcal{O}(w^{1/6})$ for the current block width w.

We modify the original data structure in two ways. First, we extend the column and row permutations with precomputed queries for rectangle point counts and cumulative cell offsets within individual fine columns/rows; this requires only sublinear additional space in the precomputed tables. Second, we build recursive column/row structures G'_C, G'_R of $\mathcal{O}(\lg \lg n)$ levels, where each node precomputes queries for cell emptiness, cell rank within a subcolumn, and rectangle point counts over its children, all fitting in $o(n)$ bits total (this is possible due to the low branching factor). A rectangle counting query decomposes R layer by layer through the hierarchy, spending $\mathcal{O}(1)$ time per level for a total of $\mathcal{O}(\lg \lg n)$. Due to the space constraints, the full construction (proving Theorem 3) is included in the full version of the paper.

4 Bounded treewidth case

In this section, we give the main idea underlying the succinct data strucure for permutations of bounded treewidth, establishing Theorem 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 5. 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 5, with the additional property that components of the division are of bounded size. This is captured in the following result, with proof in the full version.

▶ **Lemma 6.** *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 $\mathcal{O}_t(n + \sum_{i=1}^{\ell} m_i)$.

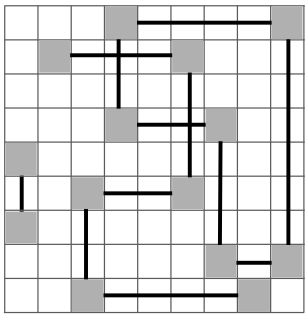

Figure 5 Three components of a division; non-zero cells shaded gray, edges shown as thick lines.

Using Lemma 6, the data structure for storing a permutation simplifies. Instead of the earlier separate row- and column- structures, we now build a single recursive structure based on the components. Assuming bounded treewidth, components are 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.

More concretely, each component of the fine division $(\mathcal{R}_2, \mathcal{C}_2)$ is stored as a *gridded permutation*: the permutation of the input points inside the component, partitioned according to the (at most $\mathcal{O}(t)$) non-zero cells it occupies, split both horizontally and vertically. This is in contrast to the column- and row permutations of § 3, which are split only one way. Consequently, from a column offset within the component we read off both the non-zero cell where the queried point falls and its exact vertical offset within the corresponding row; the symmetric query recovers a column offset from row-information. Since a component occupies $\mathcal{O}(t)$ cells and every column of $\mathcal{C}_2$ has width $\mathcal{O}(\sqrt{\lg n})$, the number of distinct gridded permutations of total width w is at most $s_\pi^w \cdot \sqrt{\lg n}^{\mathcal{O}(t)}$, so an index into a global precomputed table of them takes $w \lg s_\pi + \mathcal{O}(\lg \lg n)$ bits. As no two components share a column of $\mathcal{C}_2$ – otherwise they would be merged – these widths sum to n, giving $n \lg s_\pi + o(n)$ bits overall, while the table itself occupies $o(n)$ bits.

The remaining parts mirror § 3. The single recursive structure, which we denote G, is a three-level tree whose level-1 and level-2 nodes are the components of $(\mathcal{R}_1, \mathcal{C}_1)$, resp. $(\mathcal{R}_2, \mathcal{C}_2)$, ordered by containment, supporting subComponent navigation together with the rowRank and colRank queries. A rank query follows the same path as in § 3: the column tree T_C and index I_C supply the relative column ranks at both levels and the offset within the fine column; descending G then locates the component containing the query, whose gridded permutation resolves this offset into a non-zero cell and its within-row offset, while the accompanying rowRank queries return the relative row ranks. These ranks in turn drive the row tree T_R and index I_R to recover the absolute row position. Unrank queries are symmetric. The saving of a factor two over Theorem 1 comes from the fact that a component fully spans its own rows and columns, so all non-zero cells sharing a row or column with the queried cell lie in the same component, and collapsing the component to a single permutation loses no rank information. A single structure therefore suffices, whereas in the general case a column permutation exposes only the rank within a cell, requiring a separate row permutation.

We remark in passing that the notion of component and the decomposition in Lemma 6 relate to the concept of *component twin-width* [12]. Loosely speaking, we compute a *balanced* component twin-width decomposition, analogously to how Lemma 5 achieves a balanced twin-width decomposition. Due to space constraints, the detailed description of the data structure can be found in the full version.

References

1. Shlomo Ahal and Yuri Rabinovich. On complexity of the subpattern problem. *SIAM J. Discret. Math.*, 22(2):629–649, 2008. doi:10.1137/S0895480104444776.
2. 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.
3. 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.
4. Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theor. Comput. Sci.*, 513:109–123, 2013. doi:10.1016/J.TCS.2013.10.019.
5. Omri Ben-Eliezer, Slobodan Mitrović, and Pranjal Srivastava. Approximate counting of permutation patterns. *arXiv preprint arXiv:2411.04718*, 2024. doi:10.48550/arXiv.2411.04718.
6. 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. doi:10.1016/J.JALGOR.2005.08.001.
7. 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.
8. 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.
9. 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.
10. Miklós Bóna. *Combinatorics of permutations*. CRC Press, 2022.
11. Édouard 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.
12. 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.
13. 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.
14. Édouard Bonnet, Jaroslav Nešetřil, Patrice Ossona de Mendez, Sebastian Siebertz, and Stéphan Thomassé. Twin-width and permutations, 2021. arXiv:2102.06880.
15. 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.
16. 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. doi:10.4230/LIPIcs.ISAAC.2024.17.
17. 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.

18 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.

19 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.

20 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.

21 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.

22 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. doi:10.1137/1.9781611976465.136.

23 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.

24 Jacob Fox. Stanley-wilf limits are typically exponential, 2013. arXiv:1310.8378.

25 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.

26 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.

27 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.

28 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.

29 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.

30 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984. doi:10.1137/0213024.

31 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.

32 Sergey Kitaev. *Patterns in permutations and words*, volume 1. Springer, 2011. doi:10.1007/978-3-642-17333-2.

33 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.

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

35 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.

36 J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theor. Comput. Sci.*, 438:74–88, 2012. doi:10.1016/J.TCS.2012.03.005.

37 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.

38 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.

39 Michal Opler. Inapproximability of counting permutation patterns. In Sayan Bhattacharya, Danupon Nanongkai, Michael Benedikt, and Gabriele Puppis, editors, *53rd International Colloquium on Automata, Languages, and Programming, ICALP 2026, Royal Holloway, University of London, Egham, United Kingdom, July 7-10, 2026*, volume 374 of *LIPIcs*, pages 145:1–145:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026. doi:10.4230/LIPIcs.ICALP.2026.145.

40 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.

41 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. doi:10.4230/LIPIcs.STACS.2022.52.

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

43 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.