Submatrix maximum queries in Monge matrices and Monge partial matrices, and their applications[*]

Haim Kaplan[†] Shay Mozes[‡] Yahav Nussbaum[†] Micha Sharir[§]

Abstract

We describe a data structure for submatrix maximum queries in Monge matrices or Monge partial matrices, where a query specifies a contiguous submatrix of the given matrix, and its output is the maximum element of that submatrix. Our data structure for an $n \times n$ Monge matrix takes $O(n \log n)$ space, $O(n \log^2 n)$ preprocessing time, and can answer queries in $O(\log^2 n)$ time. For a Monge partial matrix the space bound and the preprocessing time both grow by the small factor $\alpha(n)$, where $\alpha(n)$ is the inverse Ackermann function. Our design exploits an interpretation of the column maxima in a Monge matrix (resp., Monge partial matrix) as an upper envelope of pseudo-lines (resp., pseudo-segments).

We give two applications for this data structure: (1) For a set of n points in a rectangle B in the plane, we build a data structure that, given a query point p, returns the largest-area empty axis-parallel rectangle contained in B and containing p, in $O(\log^4 n)$ time. The preprocessing time is $O(n\alpha(n) \log^4 n)$, and the space required is $O(n\alpha(n) \log^3 n)$. This improves substantially a previous data structure of Augustine et al. [arXiv:1004.0558] that requires quadratic space. (2) Given an n-node arbitrarily weighted planar digraph, with possibly negative edge weights, we build, in $O(n \log^2 n / \log \log n)$ time, a linear-size data structure that supports edge-weight updates and distance queries between arbitrary pairs of nodes (where the distance is minimum weight of a path in the graph between the pair of nodes), in $O(n^{2/3} \log^{5/3} n)$ time for each update and query. This improves the $O(n^{4/5} \log^{13/5} n)$-time bound of Fakcharoenphol and Rao [JCSS 72, 2006]. Our data structure has already been applied in a recent maximum flow algorithm for planar graphs of Borradaile et al. [FOCS 2011], and we believe it will find additional applications.

1 Introduction.

A matrix M is a *Monge matrix* if for every pair of rows $i < j$ and every pair of columns $k < \ell$ we have $M_{ik} + M_{j\ell} \leq M_{i\ell} + M_{jk}$; it is called an *inverse Monge matrix* if the reverse inequality $M_{ik} + M_{j\ell} \geq M_{i\ell} + M_{jk}$ holds for every such quadruple of indices.[1] Suppose we have two ordered sets of points, A and B, on two opposite sides of a rectangle, respectively. Monge [34] observed that for $i < j$ and $k < \ell$ any monotone path from point i of A to point ℓ of B must cross any such path from point j of A to point k of B. It follows that the matrix in which the (i, k) entry stores the distance from point i of A to point k of B has the Monge property. Consequently, Monge matrices proved to be useful in solving problems related to distances between points in the plane.

In addition, Monge matrices have many applications in combinatorial optimization and computational geometry: The traveling salesman problem can be solved in linear time if the underlying cost matrix is a Monge matrix [40]. The greedy algorithm solves the trasportation problem optimally if the costs form a Monge matrix [24]. Monge matrices were also used to obtain efficient algorithms for several problems on n-gons, like finding the k furthest vertices from any vertex [2], finding a minimum-area circumscribing d-gon, and others [2, 3]. For a survey on Monge matrices and their uses in combinatorial optimization see [11].

A particularly famous result with many applications is an algorithm by Aggarwal et al. [2] to find the minimum or the maximum in each row of an $m \times n$ Monge matrix in $O(m + n)$ time. This algorithm is known as the SMAWK algorithm for the initials of its inventors. There are also algorithms with slightly worse asymptotic running times for finding row maxima and row minima in specific kinds of Monge *partial*

[*]Work by Haim Kaplan and Yahav Nussbaum were partially supported by Grant 2006/204 from the U.S.–Israel Binational Science Foundation, and by Grant 822/10 from the Israel Science Fund. Work by Micha Sharir was partially supported by NSF Grant CCR-08-30272, by Grant 2006/194 from the U.S.–Israel Binational Science Foundation, by Grant 338/09 from the Israel Science Fund, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University. Work By Shay Mozes was partially supported by NSF Grant CCF-0964037 and by a Kanellakis fellowship.

[†]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E-mail: haimk@post.tau.ac.il, yahav.nussbaum@cs.tau.ac.il

[‡]Department of Computer Science, Brown University. E-mail: shay@cs.brown.edu

[§]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel, and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA. E-mail: michas@post.tau.ac.il

[1]In what follows we will sometimes make no distinction between Monge and inverse Monge matrices. Note that by reversing the order of the rows or of the columns, an inverse Monge matrix becomes a Monge matrix.

matrices in which some of the entries are undefined. In these cases we require that the matrix have the Monge property only with respect to the defined entries [1, 28, 29]. We note that all of these algorithms actually assume the weaker property of *total monotonicity* of the underlying matrix (see below for the definition), which is implied by the inverse Monge property.

1.1 Our contributions. We present a data structure for efficient *submatrix maximum* queries in an $n \times n$ Monge matrix.[2] Our data structure is constructed in $O(n \log^2 n)$ time, its size is $O(n \log n)$, and it can find the maximum in any (contiguous) submatrix, specified by a range of rows and a range of columns, in $O(\log^2 n)$ time.

There are two special cases for which we give more efficient data structures. (1) When the query submatrix is a *row-interval*, that is, a contiguous portion of a single row, we present a data structure that requires $O(n \log n)$ preprocessing time and $O(n \log n)$ space, and can answer queries in $O(\log n)$ time. (2) When the query submatrix is a *slab*, that is, a contiguous subsequence of complete rows, we present a data structure that can be constructed in $O(n)$ time and can answer a query in $O(1)$ time.

We show how to extend our data structures to Monge *partial* matrices, which are Monge matrices that contain undefined entries, so that the defined entries form within each row or column a contiguous interval. The query times of the submatrix maximum, row interval maximum, and slab maximum data structures all remain the same, but their sizes and construction times grow by a factor of $\alpha(n)$, except for the construction time of the row-interval data structure which grows by the slightly larger factor $\alpha(n) \log n$.

We obtain these data structures using techniques from computational geometry. Specifically, we think of the rows of the matrix as functions over the discrete sequence of column indices, and exploit the fact that the Monge property implies that these functions are in fact discrete variants of pseudo-lines (or pseudo-segments in case of partial matrices). This connection may prove useful for constructing other data structures for Monge matrices. We describe our data structures in Section 3.

Our Monge row-interval minimum data structure has already been applied in a recent maximum flow algorithm for planar graphs of Borradaile et al. [10] (involving a subset of the authors). In this paper we present two new applications of our data structures. In the first application we give, for a planar point set enclosed in some fixed axis-parallel box B, an almost linear-size structure for finding the largest-area empty axis-parallel rectangle containing a query point, where the previous best solution required quadratic space. In the second application we substantially improve a data structure of Fakcharoenphol and Rao [20] for distance queries in a dynamically weighted planar digraph when the edge weights can be negative. We elaborate on these two applications next.

Finding the largest empty rectangle containing a query point. Let P be a set of n points in a fixed axis-parallel rectangle B in the plane. A *P-empty rectangle* (or just an empty rectangle for short) is any axis-parallel rectangle that is contained in B and its interior does not contain any point of P. We consider the problem of preprocessing P into a data structure so that, given a query point q, we can efficiently find the largest-area P-empty rectangle containing q. This problem arises in electronic design automation, in the context of the design and verification of physical layouts of integrated circuits (see, e.g., [42, Chapter 9]).

The largest-area P-empty rectangle containing q is a *maximal empty rectangle*, namely, it is a P-empty rectangle not contained in any other P-empty rectangle. Each side of a maximal empty rectangle abuts a point of P or an edge of B. See Figure 1 for an illustration. Maximal empty rectangles arise in the enumeration of "maximal white rectangles" in image segmentation [7].

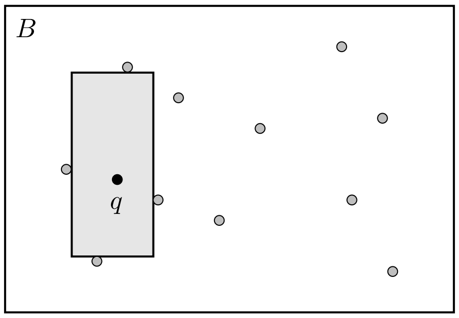

Figure 1: A maximal P-empty rectangle containing a query point q.

Augustine et al. [6] gave a data structure for this problem whose storage and preprocessing time are both $O(n^2 \log n)$, and the query time is $O(\log n)$. In Section 4, we significantly improve this result, in terms of storage and preprocessing time. Specifically, we present a data structure that requires $O(n\alpha(n) \log^3 n)$ space and can answer a query in $O(\log^4 n)$ time. The structure can be constructed in $O(n\alpha(n) \log^4 n)$ time.

In a nutshell, our algorithm computes all the maximal P-empty rectangles and preprocesses them into a

[2]To simplify the presentation we discuss square matrices here. Our results apply also to rectangular matrices, see Section 3.

data structure which is then searched with the query point. A major problem that one faces is that the number of maximal P-empty rectangles can be quadratic in n (see, e.g., Figure 5), so we cannot afford to compute them explicitly. Instead, we exploit the inverse Monge partial matrix structure that exists in certain configurations, which facilitates a faster search for the maximum. The inverse Monge property of areas of certain configurations of rectangles has been already observed in McKenna et al. [33] and used by Aggarwal and Suri [4] for finding the largest (global) P-empty rectangle.

Our data structure can be modified to find the largest-perimeter P-empty rectangle containing a given query point q, in a way similar to the algorithm of [4].

Dynamic shortest path queries with negative edge weights in planar graphs. Let G be an n-node weighted directed planar graph, where the weights of the edges of G are arbitrary real numbers, possibly negative. In Section 5, we present a dynamic data structure that can answer distance queries (that is, return the minimal path weight) between arbitrary pairs of nodes in G, and allows updates of edge weights. The time for a query or update is $O(n^{2/3} \log^{5/3} n)$. The construction time of our data structure is $O(n \log^2 n / \log \log n)$, and it requires linear space.

Our data structure is based on the data structure of Fakcharoenphol and Rao [20], which has $O(n^{4/5} \log^{13/5} n)$ query time and requires $O(n \log n)$ space, and on the data structure of Klein [30], which has the same query time and space bound as our algorithm but does not support negative edge weights.

Italiano et al. [26] extended the data structure of Klein to allow insertions and deletions of edges that do not change the embedding of the graph. This technique also applies to our data structure, and we can extend it to support insertions and deletions of edges, retaining the same asymptotic time bounds for updates (including insertions, deletions, and changes of edge weights) and queries.

1.2 Related work. The problem of finding the largest empty rectangle containing a query point was introduced in the paper mentioned above by Augustine et al. [6]. An easier problem that has been studied more extensively is that of finding the largest-area P-empty axis-parallel rectangle contained in B. Notice that the largest P-empty *square* is easier to compute, because its center is a Voronoi vertex in the L_∞-Voronoi diagram of P (and of the edges of B), which can be found in $O(n \log n)$ time [17, 32]. There have been several studies on finding the largest-area maximal empty rectangle [5, 14, 37] in B; the fastest algorithm to date, by Aggarwal and Suri [4], takes $O(n \log^2 n)$ time and $O(n)$ space. We use many of the observations in these algorithms. Most important of which is the quadratic number of maximal rectangles generated by two "parallel" monotonic chains of points, and the inverse Monge property which they satisfy. Our main contribution is adapting these ideas to construct a *data structure* for finding the largest empty rectangle containing a query point.

Nandy et al. [38] show how to find the largest-area axis-parallel empty rectangle avoiding a set of polygonal obstacles, within the same time bounds. Boland and Urrutia [9] present an algorithm for finding the largest-area axis-parallel rectangle inside an n-sided simple polygon in $O(n \log n)$ time. Chaudhuri et al. [13] give an algorithm to find the largest-area P-empty rectangle, with no restriction on its orientation, in $O(n^3)$ time.

The problem can be studied for regions other than axis-parallel rectangles. Augustine et al. [6] studied the case where the regions containing the query point are disks. For this case they gave a data structure that requires $O(n^2)$ space, $O(n^2 \log n)$ preprocessing time, and can answer a query in $O(\log n)$ time.

Static data structures that preprocess an input planar graph to answer distance queries efficiently were studied by many authors [12, 16, 19, 20, 21, 22, 30, 35, 39]. The dynamic setting in which updates of edge weights are supported was recently studied by Fakcharoenphol and Rao [20]. They describe a data structure that supports only non-negative edge weights, requires $O(n \log n)$ space, $O(n \log^3 n)$ preprocessing time, and performs a query or update in $O(n^{2/3} \log^{7/3} n)$ time. They also gave a data structure which supports negative edge weights but with query and update time of $O(n^{4/15} \log^{13/5} n)$, with the same space and preprocessing time. Klein [30], using a technique of Fakcharoenphol and Rao, gave a simpler data structure for the case of non-negative edge weights that requires linear size, $O(n \log n)$ preprocessing time, and $O(n^{2/3} \log^{5/3} n)$ time per query and update. Our data structure is based on these results of Fakcharoenphol and Rao and of Klein.

2 Preliminaries.

Totally monotone and Monge matrices. A matrix M is *totally monotone* if, for every pair of rows $i < j$ and every pair of columns $k < \ell$, $M_{ik} \leq M_{i\ell}$ implies $M_{jk} \leq M_{j\ell}$. The SMAWK algorithm of Aggarwal et al. [2] finds all row maxima in a totally monotone $m \times n$ matrix in $O(m + n)$ time. By negating the entries of the matrix and reversing the order of the columns to recover total monotonicity, we obtain a totally monotone matrix in which the maximum of each row is the negation of the minimum entry of the row in the original matrix. By applying the SMAWK algorithm to the transformed matrix we can also find the row minima in a totally monotone $m \times n$ matrix in $O(m + n)$ time.

A matrix M is a *Monge matrix* (also called *concave Monge* matrix) if for every pair of rows $i < j$ and every pair of columns $k < \ell$ we have $M_{ik} + M_{j\ell} \leq M_{i\ell} + M_{jk}$. A matrix M is an *inverse Monge matrix* (also called *convex Monge* matrix) if for every pair of rows $i < j$ and every pair of columns $k < \ell$ we have $M_{ik} + M_{j\ell} \geq M_{i\ell} + M_{jk}$. Clearly if M is Monge (resp., inverse Monge) then so is its transpose M^t. It is easy to verify that if M is an inverse Monge matrix, then M and M^t are totally monotone. Also, if M is a Monge matrix, then by negating the entries of M, or by reversing the order of the rows or of the columns, we get an inverse Monge matrix. Therefore, we can use the SMAWK algorithm for finding row maxima, row minima, column maxima and column minima in a Monge matrix or in an inverse Monge matrix.

We say that a matrix M is a *partial matrix* if some of the entries of M are undefined, such that the defined entries of each row are consecutive, and the defined entries of each column are consecutive.[3] See Figure 2. A totally monotone (Monge, inverse Monge) partial matrix is a partial matrix whose defined entries satisfy the total monotonicity (resp., Monge, inverse Monge) condition.

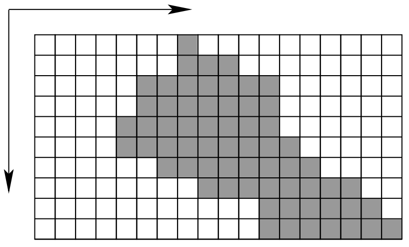

Figure 2: The defined part of a partial matrix.

Upper envelopes of pseudo-lines and of pseudo-segments. A set L of m x-monotone unbounded (resp., bounded) Jordan curves in the plane is called a family of *pseudo-lines* (resp., *pseudo-segments*) if every pair of curves intersect in at most one point, and the two curves cross each other there.

We think of a pseudo-line ℓ as a function $\ell(x), x \in \mathbb{R}$, and of a pseudo-segment s as a function $s(x), x \in I_s$, where $I_s \subseteq \mathbb{R}$ is some (possibly infinite) interval. The

[3]Aggarwal and Klawe [1] use a different definition of partial matrices in a similar context. The two definitions, however, are nearly equivalent; any partial matrix of [1] can be decomposed into two partial matrices as defined in this paper, and vice versa.

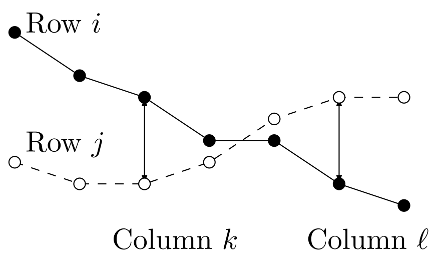

Figure 3: The pseudo-lines of two rows in an inverse Monge matrix.

upper envelope of a set of pseudo-lines L is the function $\mathcal{E}_L(x) = \max_{\ell \in L} \ell(x)$. The *upper envelope* of a set S of pseudo-segments is the function $\mathcal{E}_S(x)$ defined as follows: $\mathcal{E}_S(x) = \max_{s \in S}\{s(x) \mid x \in I_s\}$ if there is an interval I_s for some $s \in S$ which contains x, and $\mathcal{E}_S(x) = -\infty$ otherwise.

A *breakpoint* in the upper envelope $\mathcal{E}_L(x)$ of a set of pseudo-lines L is an intersection point of two pseudo-lines on $\mathcal{E}_L(x)$. A *breakpoint* in the upper envelope $\mathcal{E}_S(x)$ of a set S of pseudo-segment is either an intersection point of two pseudo-segments or an endpoint of one of the pseudo-segments. We define the *complexity* of an envelope $\mathcal{E}_L(x)$ or $\mathcal{E}_S(x)$ to be the number of its breakpoints. Since each pseudo-line in L can appear along the upper envelope in a single connected (possibly empty) interval, the complexity of $\mathcal{E}_L(x)$ is $O(|L|)$. The complexity of $\mathcal{E}_S(x)$ is known to be $O(|S|\alpha(|S|))$ [41].

Monge matrices and pseudo-lines. Let M be an $m \times n$ inverse Monge matrix. We can think of the entries of a particular row ρ as defining a (discrete) function $\hat{M}_\rho(\cdot)$, mapping each index π of a column to the value of $M_{\rho\pi}$. By the total monotonicity of M^t we get that $M_{ik} \leq M_{jk}$ implies $M_{i\ell} \leq M_{j\ell}$, for any four indices $i < j$ and $k < \ell$, and this in turn implies that these functions behave as pseudo-lines. Specifically, we extend the domain of definition of each function $\hat{M}_\rho$ to the continuous interval $[1, n]$, by linearly interpolating between each pair of π-consecutive points. Then each pair of the resulting piecewise linear functions intersect at most once. Moreover, the pseudo-line of row i lies above the pseudo-line of row j to the left of their intersection point, and this order is reversed to the right of that point; very informally, the "slope" of the pseudo-line of row i is smaller than that of the pseudo-line of row j. See Figure 3.

Similarly if M is an $m \times n$ inverse Monge partial matrix then, since the defined entries in each row are consecutive, we can think of each row as a discrete partially defined function, which, after interpolating the functions in the same manner as above, is defined over some connected subinterval of $[1, n]$. By the total monotonicity of M^t we get that $M_{ik} \leq M_{jk}$ implies $M_{i\ell} \leq M_{j\ell}$, when $i < j$ and $k < \ell$ and these four entries of M are all defined. Therefore the interpolated partial functions corresponding to the rows of M form a family of pseudo-segments.

We note that the preceding analysis also applies to any matrix whose transpose is a totally monotone matrix.

3 The data structure.

In this section we develop the main result of our paper, data structures for submatrix maximum queries in (inverse) Monge matrices and in (inverse) Monge partial matrices.

The model that we assume, which is the same model used by all the previous studies mentioned in the introduction, is that the input matrix M is not given explicity—it has too many entries. Instead, it is given implicitly so that one can retrieve any desired entry M_{ij} of M in $O(1)$ time.

3.1 Submatrix maximum in Monge matrices.

Let M be an $m \times n$ inverse Monge matrix and consider the interpretation of the rows of M as pseudo-lines (see Section 2). The upper envelope of the rows of M consists of the column maxima of M. An explicit representation of the entire upper envelope requires $O(n)$ values. However, the envelope contains only $O(m)$ breakpoints, and we use these breakpoints to implicitly represent the envelope.[4] We will use this compact representation for upper envelopes of several subsets of the rows, and we note that the saving becomes significant when the number of rows is significantly smaller than the number of columns.

Every column π is contained in some *interval* between two consecutive breakpoints of the upper envelope, and we can find that interval using binary search on the breakpoints. Once we found the interval, we know to which pseudo-line it belongs, and we therefore know which row contains the maximum of column π and then retrieve that maximum in $O(1)$ time. We conclude that we can find the maximum in a column π of M in $O(\log m)$ time, given the implicit representation of the upper envelope of the pseudo-lines of the rows of M as a list of its breakpoints.

[4]Technically, since we are dealing with discrete versions of pseudo-lines, a breakpoint occurs in general "between" two consecutive columns. The representation of breakpoints is handled accordingly, but we will not refer to this issue explicitly in what follows.

We find the implicit representation of the upper envelope of all pseudo-lines in $O(m(\log m + \log n))$ time using the following standard divide-and-conquer approach. We build a balanced binary tree T_h over the rows of M. For each node u of T_h we compute the upper envelope of the pseudo-lines representing the rows in the subtree rooted at u (which we call the *upper envelope of u* for short). The upper envelope of a leaf is trivial, since it represents a single row, and no computation is needed. For an internal node u, we construct its upper envelope by merging the envelopes of its two children w_1 and w_2, where w_1 is the child whose rows have lower indices. Let k be the number of rows at the leaves of the subtree of T_h rooted at u. The number of breakpoints in each of the upper envelopes of u, w_1, and w_2 is $O(k)$. By the total monotonicity of M and its implications discussed above, the upper envelope of u starts with a prefix of the upper envelope of w_1, reaches a breakpoint between the pseudo-line of a row of w_1 and that of a row of w_2, and ends with a suffix of the upper envelope of w_2. We find the breakpoint between the upper envelopes of w_1 and of w_2 using binary search on the columns. For each column in the binary search, we find its maximum in the upper envelope of w_1 and in the upper envelope of w_2 in $O(\log k)$ time. Therefore the binary search for the breakpoint between the two upper envelopes takes $O(\log k \log n)$ time. Then, we construct the upper envelope of u in $O(k)$ time, by concatenating the prefix of the upper envelope of w_1, the new breakpoint, and the suffix of the upper envelope of w_2. The total time for constructing the upper envelope of u from those of its children is $O(k + \log k \log n)$, which sums to $O(m(\log m + \log n))$ over the entire tree. The total size of T_h is $O(m \log m)$.

We use the tree T_h to create a data structure for reporting the maximum of a column within a given range of consecutive rows. A query in this data structure is a column π and a range of rows $[\rho, \rho']$, and the output is the maximum entry of M at column π between rows ρ and ρ' (inclusive). We answer such a query as follows. There are $O(\log m)$ *canonical nodes* of T_h whose sets of rows are disjoint and cover $[\rho, \rho']$. (The set of rows of each such node u is contained in $[\rho, \rho']$, but the set of rows of the parent of u is not.) For each such canonical node u, we locate the interval of the envelope of u containing π, and hence the maximum of column π among the rows of u. The output is the largest of these $O(\log m)$ maxima. A binary search within each envelope takes $O(\log m)$ time, and therefore the total query time is $O(\log^2 m)$.

We can reduce the query time by a logarithmic factor using fractional cascading [15]. This technique allows us to insert bridges from the envelope of a node

u of T_h to the envelopes of its two children, such that once we locate the interval containing column π in the envelope of u, we can locate the interval containing column π in the envelope of its children in $O(1)$ time. This construction does not incur any space or time overhead, but reduces the query time to $O(\log m)$.

Note that for this data structure we only used the fact that M^t is a totally monotone matrix (see a comment made earlier). Therefore, by transposing the matrix, we get the *row-interval maximum* data structure for a totally monotone matrix:

LEMMA 3.1. *Given a totally monotone matrix of size $m \times n$, one can construct, in $O(n(\log m + \log n))$ time, a data structure of size $O(n \log n)$ that can report the maximum entry in a query row and a range of columns in $O(\log n)$ time.*

We note that by a symmetric treatment of the lower envelope of pseudo-lines we can construct a variant of the data structure that reports minima rather than maxima.

We continue now to construct a data structure that answers maximum queries within a submatrix of M. We build the tree T_h over the rows of M with an upper envelope for each node of T_h in $O(m(\log m + \log n))$ time as before. We also construct, in $O(n(\log m + \log n))$ time, the "flipped" row interval maximum data structure of Lemma 3.1 for finding the maximum element of a row within a consecutive range of columns, and denote it by $\mathcal{B}$; queries in $\mathcal{B}$ take $O(\log n)$ time. Let u be a node of T_h. We find and store the maximum in every interval of the upper envelope of u by an appropriate query to $\mathcal{B}$. There are $O(m \log m)$ such intervals in all nodes of T_h, so finding their maxima takes a total of $O(m \log m \log n)$ time, which is the bottleneck step of the construction. For every node u of T_h we build a range maximum query data structure over the maxima of the intervals of the upper envelope of u. For our purpose it is sufficient to use a binary search tree over these intervals, augmented with subtree maxima stored at its nodes, which can answer a range maximum query in $O(\log m)$ time. Later on, in Section 3.3, we will use the more sophisticated structure of [8] for another variant of our data structure.

A query in this data structure is a submatrix of M with specified ranges R of consecutive rows and C of consecutive columns, and the answer is the maximum entry in this submatrix. To answer such a query, we first represent R as the (disjoint) union of $O(\log m)$ subtrees of T_h. For each root u of such a subtree, we find the maximum of the upper envelope of u in the range defined by C as follows; refer to Figure 4. We find the maximum set $\mathcal{I}$ of intervals of the upper envelope of u which are fully contained in C. Then we find the maximum in the range spanned by $\mathcal{I}$ using the range maximum data structure that we have built over the intervals of the envelope of u. The prefix p of C which is not covered by $\mathcal{I}$ is contained in a single interval of the upper envelope of u. Therefore the maximum of the upper envelope of u within p is in some single row ρ. Similarly the maximum of the upper envelope of u in the suffix s of C not covered by $\mathcal{I}$ is obtained in some single row ρ'. We use the structure $\mathcal{B}$ to find the maximum in row ρ within column range p, and the maximum in row ρ' within column range s. This way we cover the entire query submatrix. The query time is $O(\log m(\log m + \log n))$, since we need to retrieve and to search in $O(\log m)$ canonical nodes u and we find the maximum in the upper envelope of each such u in $O(\log m + \log n)$ time. We thus obtain the following *submatrix maximum* data structure:

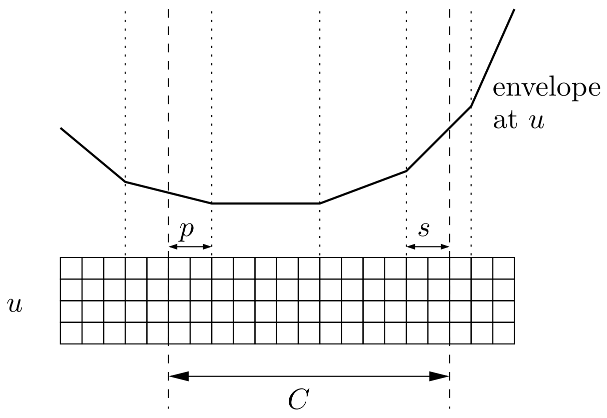

Figure 4: Maximum query at a single node u of T_h.

LEMMA 3.2. *Given an inverse Monge matrix of size $m \times n$, one can construct in $O(m \log m \log n + n(\log m + \log n))$ time a data structure of size $O(m \log m)$ that can report the maximum entry in a given submatrix in $O(\log m(\log m + \log n))$ time.*

Again, we can construct a variant of this data structure for finding minima instead of maxima within the same time bounds. The same result also applies to Monge matrices.

3.2 Submatrix maximum in Monge partial matrices.

We extend the two data structures from the previous section to inverse Monge partial matrices. Let M denote an $m \times n$ inverse Monge partial matrix, and consider the interpretation of the rows of M as pseudo-segments. Since the complexity of the upper envelope of k pseudo-segments is $O(k\alpha(k))$ [41], it follows that there are $O(k\alpha(k))$ breakpoints in the upper envelope of any subset of k rows.

We use again a divide-and-conquer approach, but this time merging the upper envelopes of the two children w_1, w_2 of a node u of T_h is slightly more involved, since the envelopes may cross each other multiple times. To merge the envelopes, we first merge the lists of breakpoints representing the envelopes of w_1 and w_2. We traverse the merged list of endpoints from left to right, and at each breakpoint we compare the rows that contain the intervals of the two upper envelopes at the column of the breakpoint. We add the breakpoint to the upper envelope of u, only if it remains on this envelope. When we get two consecutive breakpoints that came from two different upper envelopes, there must be a new breakpoint between them. We find the new breakpoint in $O(\log n)$ time using a binary search on the columns of the current intervals in the two upper envelopes. Since we pay $O(\log n)$ time to find a new breakpoint, and there are $O(m\alpha(m)\log m)$ breakpoints in total, the construction takes $O(m\alpha(m)\log m \log n)$ time, and the required space is $O(m\alpha(m)\log m)$. The query time (with fractional cascading) remains $O(\log m)$.

An alternative approach is to apply Hershberger's algorithm [23], which constructs the upper envelope of k segments in $O(k \log k)$ time. The algorithm is also applicable to the case at hand of pseudo-segments, and, as above, we incur the extra factor $O(\log n)$ for finding the intersection of two pseudo-segments. However, since we need to construct an entire hierarchy of envelopes, Hershberger's algorithm does not make the whole procedure more efficient.

Again, by applying the algorithm described above to M^t, we get the following row-interval maximum data structure:

LEMMA 3.3. *Given an totally monotone partial matrix of size $m \times n$, one can construct, in $O(n\alpha(n)\log m \log n)$ time, a data structure of size $O(n\alpha(n)\log n)$ that can report the maximum entry in a query row and a contiguous range of columns in $O(\log n)$ time.*

We next develop the submatrix maximum data structure. Similar to what was done before, we construct the tree T_h and the row maxima data structure $\mathcal{B}$, but this time the construction takes $O((m\alpha(m) + n\alpha(n))\log m \log n)$ time. Since the overall number of breakpoints in the upper envelopes is now $O(m\alpha(m)\log m)$, the total time for finding the maximum in every interval between two consecutive breakpoints, over all envelopes, is now $O(m\alpha(m)\log m \log n)$. Therefore, the total construction time is $O((m\alpha(m) + n\alpha(n))\log m \log n)$, and the total size for the data structure is $O(m\alpha(m)\log m)$. The rest of the data structure is constructed exactly as in the case of full matrices, and the query time remains $O(\log m(\log m + \log n))$.

In conclusion, we get the following submatrix maximum data structure for partial matrices:

LEMMA 3.4. *Given an inverse Monge partial matrix of size $m \times n$, one can construct, in $O((m\alpha(m) + n\alpha(n))\log m \log n)$ time, a data structure of size $O(m\alpha(m)\log m)$ that can report the maximum entry in a query submatrix in $O(\log m(\log m + \log n))$ time.*

Again, the same results apply for Monge partial matrices. As in Section 3.1, we can also apply symmetric variants of the above constructions for answering submatrix minimum queries.

3.3 Maximum in a consecutive range of rows.

In the special case where the query submatrices consist of contiguous ranges of complete rows (i.e., with the entire range of columns), we can produce a more efficient *slab maximum* data structure. Specifically, we find the maximum in each row using the SMAWK algorithm [2], and then construct in linear time a data structure that answers range maximum queries on contiguous ranges of the row maxima, in $O(1)$ time; see [8] and the references therein.

LEMMA 3.5. *Given a totally monotone matrix of size $m \times n$, one can construct, in $O(m + n)$ time, a data structure of size $O(m)$ that can report the maximum entry in a query set of complete consecutive rows, in $O(1)$ time.*

For the case of Monge partial matrices we can construct a similar data structure that uses the algorithm of Klawe and Kleitman [29] instead of the SMAWK algorithm.[5] This yields:

LEMMA 3.6. *Given a totally monotone partial matrix of size $m \times n$, one can construct, in $O(n\alpha(m) + m)$ time, a data structure of size $O(m)$ that can report the maximum entry in a query range of consecutive rows, in $O(1)$ time.*

4 Maximal empty rectangle containing a query point.

Let P be a set of n points inside an axis-parallel rectangular region B in the plane. In this section we present the first application of our data structures,

[5]The definition of partial matrices in [29] is the one in [1], which is different than the definition used in this paper. However, the algorithm of [29] does apply in our case since, as previously noted, we can decompose a partial matrix into two partial matrices in the sense of [29] and run the algorithm of [29] on each of the two matrices.

which is an algorithm that preprocesses P into a data structure, so that, given a query point $q \in B$, we can efficiently find the largest-area axis-parallel P-empty rectangle containing q and contained in B.

We assume that the points of P are in general position, so that (i) no two points have the same x-coordinate or the same y-coordinate, and (ii) all the maximal P-empty rectangles have distinct areas.

One of the auxiliary structures that we use is a two-dimensional segment tree, which stores certain subsets of P-maximal empty rectangles. Here is a brief review of the structure, provided for the sake of completeness. Let $\mathcal{M}$ be a set of N axis-parallel rectangles in the plane. We first construct a standard segment tree S [18] on the x-projections of the rectangles in $\mathcal{M}$. This is a balanced binary search tree whose leaves correspond to the intervals between the endpoints of the x-projections of the rectangles. The *span* of a node v is the minimal interval containing all intervals corresponding to the leaves of its subtree. We store a rectangle R at each node v such that the x-projection of R contains the span of v but does not contain the span of the parent of v. The tree has $O(N)$ nodes, each rectangle is stored at $O(\log N)$ nodes, and the size of the structure is thus $O(N \log N)$. All the rectangles containing a query point q must be stored at the nodes on the search path of the x-coordinate of q in the tree.

For each node u of S we take the set $\mathcal{M}_u$ of rectangles stored at u, and construct a secondary segment tree S_u, storing the y-projections of the rectangles of $\mathcal{M}_u$. The total size and the preprocessing time of the resulting two-dimensional segment tree is $O(N \log^2 N)$. We can retrieve all rectangles containing a query point q by traversing the search path π of (the x-coordinate of) q in the primary tree, and then by traversing the search paths of (the y-coordinate of) q in each of the secondary trees associated with the nodes along π. The rectangles stored at the secondary nodes along these paths are exactly those that contain q. If we store at each secondary node only the rectangle of largest area among those assigned to that node, we can easily find the largest-area rectangle of $\mathcal{M}$ containing a query point, in time $O(\log^2 N)$. Storing only one rectangle at each secondary node reduces the size of the segment tree to $O(N \log N)$, but the preprocessing time remains $O(N \log^2 N)$.

This simple-minded solution will be efficient only when the size N of $\mathcal{M}$ is linear or nearly linear in n. Unfortunately, as already noted, in general the number of maximal empty rectangles can be quadratic in the input size, so for most of them we will need an additional, implicit representation. For this purpose we will decompose our problem into subproblems so that in each of the subproblems most of the maximal empty rectangles can be represented in an inverse Monge partial matrix, and we will use our submatrix maximum query data structure on the resulting matrix.

Naamad et al. [37] classified the maximal P-empty rectangles within B according to the number of their edges that touch the edges of B. They showed that there are only $O(n)$ maximal P-empty rectangles with at least one edge on ∂B, and that we can compute these rectangles in $O(n \log n)$ time. We precompute these rectangles and store them in a two-dimensional segment tree S, as described above. At query time we find the rectangle of largest area among those special "anchored" rectangles that contain the query point q, in $O(\log^2 n)$ time, by searching with q in S. (The segment tree S will also store additional rectangles that will arise in later steps of the construction; see below for details.)

In the remainder of the section we are concerned only with maximal P-empty rectangles supported by four points of P, one on each side of the rectangle. We refer to such rectangles as *bounded* P-empty rectangles. We note that the number of such rectangles can be $\Theta(n^2)$ in the worst case [37]; see Figure 5 for an illustration of the lower bound. The upper bound follows by observing that there is at most one maximal P-empty rectangle whose top and bottom edges pass through two respective specific points of P. (To see this, take the rectangle having these points as a pair of opposite vertices and, assuming it to be P-empty, expand it to the left and to the right until its left and right edges hit two additional respective points.) Handling these (potentially quadratically many) rectangles has to be done implicitly, in a manner that we now proceed to describe.

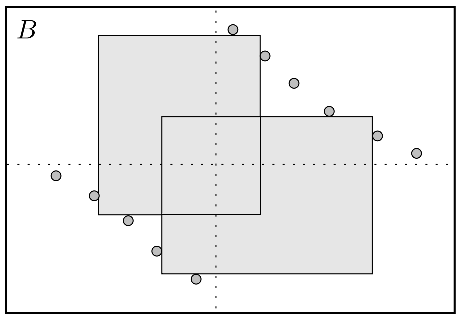

Figure 5: A set P of n points with $\Theta(n^2)$ maximal P-empty rectangles.

4.1 Maximal empty rectangles supported by four points of P.

We store the points of P in a two-dimensional range tree (see, e.g., [18]). The points are stored at the leaves of the primary tree T in their left-to-right order. For a node u of T, we denote by P_u

the subset of the points stored at the leaves of the subtree rooted at u. We associate with each internal node u of T a *vertical splitter* ℓ_u, which is a vertical line separating the points stored at the left subtree of u from those stored at the right subtree. These splitters induce a hierarchical binary decomposition of the plane into vertical strips. The strip σ_{root} associated with the root is the entire plane, and the strip σ_u of a node u is the portion of the strip of the parent $p(u)$ of u which is delimited by $\ell_{p(u)}$ and contains P_u.

With each node u in T we associate a secondary tree T_u containing the points of P_u in a bottom-to-top order. For a node v of T_u, we denote by P_v the points stored at the leaves of the subtree rooted at v. We associate with each internal node v of T_u a *horizontal splitter* ℓ_v, which is a horizontal line separating the points stored at the left subtree of v from those stored at the right subtree. These splitters induce a hierarchical binary decomposition of the strip σ_u into rectangles. The rectangle associated with the root of T_u is the entire vertical strip σ_u, and the rectangle B_v of a node v is the portion of the rectangle of the parent $p(v)$ of v which is delimited by $\ell_{p(v)}$ and contains P_v. See Figure 6.

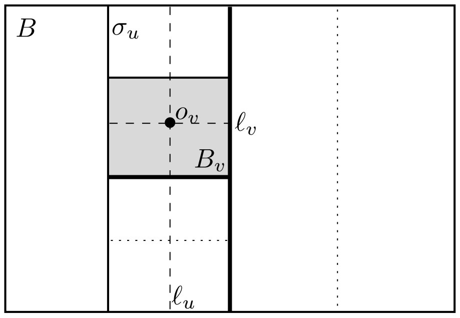

Figure 6: The rectangle B_v associated with a secondary node v, with its splitters and origin.

In this way, the range tree defines a hierarchical subdivision of the plane, so that each secondary node v is associated with a rectangular region B_v of the subdivision. If v is not a leaf then it is associated with a horizontal splitter ℓ_v. If the primary node u associated with the secondary tree of v is also not a leaf then v is also associated with a vertical splitter ℓ_u. The vertical splitter ℓ_u and the horizontal splitter ℓ_v meet at a point o_v inside B_v, which we refer to as the *origin of* v.

A query point q defines a search path π_q in T and a search path in each secondary tree T_u of a primary node u on π_q. We refer to the nodes on these $O(\log n)$ paths as constituting the *search set* of q, which therefore consists of $O(\log^2 n)$ secondary nodes.

Let R be a bounded maximal P-empty rectangle containing q supported by four points p_t, p_b, p_ℓ, and p_r of P, lying respectively on the top, bottom, left, and right edges of R. Let u be the lowest common ancestor of p_ℓ and p_r in the primary tree, and let v be the lowest common ancestor of p_t and p_b in T_u (clearly, both p_t and p_b belong to T_u). By construction, R is contained in B_v and contains both q and o_v. See Figure 7. Note that both v and u are internal nodes (each being a lowest common ancestor of two leaves) so o_v is indeed defined. Furthermore, one can easily verify that v is in the search set of q.

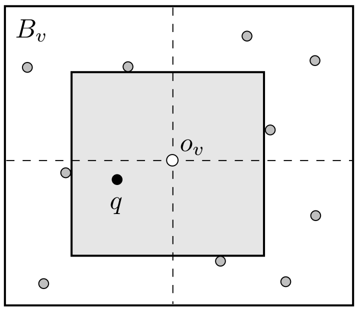

Figure 7: A bounded maximal P-empty rectangle of the subproblem at v.

In the following we consider only secondary nodes v which are not leaves, and are associated with primary nodes u which are not leaves either.

We define the *subproblem at a secondary node* v (of the above kind) as the problem of finding the largest-area bounded maximal P-empty rectangle containing q and o_v which lies in the interior of B_v. It follows that if we solve each subproblem at each secondary node v in the search set of q, and take the rectangle of largest area among those subproblem outputs, we get the largest-area bounded maximal P-empty rectangle containing q.

In the remainder of this section we focus on the solution of a single subproblem at a node v of a secondary tree T_u in the search set of q. We focus only on the points in P_v and for convenience we extend B_v to the entire plane and we move o_v to the origin. The line ℓ_u becomes the y-axis, and the line ℓ_v becomes the x-axis. Put $n_v = |P_v|$. We classify the bounded maximal P-empty rectangles contained in B_v and containing the origin according to the quadrants containing the four points associated with them, namely, those lying on their boundary, and find the largest-area rectangle containing q in each class separately.

(i) Three defining points in a halfplane. The easy cases are when one of the four halfplanes defined by the x-axis or the y-axis (originally ℓ_u and ℓ_v) contains three of the defining points. Chazelle et al. [14]

showed that there are $O(n_v)$ bounded maximal empty rectangles of this type associated with v, which we can find in $O(n_v \log n_v)$ time. Summing this cost over all secondary nodes v, we obtain a total of $O(n \log^2 n)$ such rectangles, which can be constructed in $O(n \log^3 n)$ overall time.

We add all these rectangles to the global segment tree S. The size of the expanded tree S remains $O(n \log n)$ since it still suffices to store only the largest-area rectangle among all rectangles associated with each secondary node. The preprocessing time increases to $O(n \log^4 n)$ since each of the $O(n \log^2 n)$ rectangles is mapped to $O(\log^2 n)$ secondary nodes of S, and for each rectangle R and a node u to which R is mapped, we need to check whether R is the largest rectangle mapped to u. A query in S still takes $O(\log^2 n)$ time.

The remaining cases involve bounded maximal P-empty rectangles R such that each of the four halfplanes defined by the y-axis or the x-axis contains exactly two defining points of R. This can happen in two situations: either there exist two opposite quadrants, each containing two defining points of R, or each quadrant contains exactly one defining point of R.

(ii) One defining point in each quadrant. The situation in which each quadrant contains exactly one defining point of R is also easy to handle, because again there are only $O(n_v)$ bounded maximal P-empty rectangles of this type in B_v. To see this, consider, without loss of generality, the case where the first quadrant contains the right defining point, p_r, the second quadrant contains the top defining point, p_t, the third quadrant contains the left defining point, p_ℓ, and the fourth quadrant contains the bottom defining point, p_b. See Figure 8. (There is one other situation, in which the top defining point lies in the first quadrant, the right point in the fourth quadrant, the bottom point in the third, and the left point in the second; this case is handled in a fully symmetric manner.)

We claim that p_r can be the right defining point of at most one such rectangle. Indeed, if p_r is the right defining point of such a rectangle R then p_b is the first point we hit when we sweep downwards a horizontal line segment connecting p_r to the y-axis (assuming the sweep reaches below the x-axis; otherwise p_r cannot be the right defining point of any rectangle of the current type). Similarly, the point p_ℓ is the first point that we hit when we sweep to the left a vertical line segment connecting p_b to the x-axis, p_t is the first point we hit when we sweep upwards a horizontal line segment connecting p_ℓ to the y-axis, and finally p_r is the first point we hit when we sweep a vertical line segment connecting p_t to the x-axis. As noted, if any of the points we hit during these

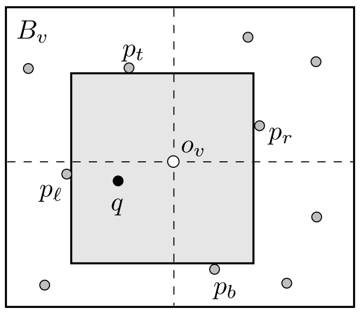

Figure 8: A bounded maximal P-empty rectangle with one defining point in each quadrant.

sweeps is not in the correct quadrant, or the last sweep does not hit p_r (e.g., because the point p_t is lower than p_r), or one of the sweeps does not hit any point before hitting ∂B_v, then p_r is not the right defining point of any rectangle of this type.

We compute these $O(n_v)$ bounded maximal P-empty rectangles using four balanced search trees. We maintain the points to the right of the y-axis in a balanced search tree Σ_r, sorted by their y coordinates, storing with each node the leftmost point in its subtree. Similarly, we maintain the points below the x-axis in a balanced search tree Σ_b sorted by their x coordinates, storing with each node the topmost point in its subtree. We maintain the points to the left of the y-axis and the points above the x-axis in symmetric search trees Σ_ℓ and Σ_t, respectively. We can find each rectangle in this family by four queries, starting with each point p_r in the first quadrant, first in Σ_b to identify p_b, then in Σ_ℓ to find p_ℓ, in Σ_t to find p_t, and finally in Σ_r to ensure that we get back to p_r.

Summing over all secondary nodes v, we have $O(n \log^2 n)$ such rectangles, which we can construct in $O(n \log^3 n)$ overall time. We add these rectangles to the global segment tree S, without changing the asymptotic bounds on its performance parameters.

4.2 Two defining points in the first and third quadrants.

The hardest case is where, say, each of the first and third quadrants contains two defining points of R. (The case where each of the second and fourth quadrants contains two defining points is handled symmetrically.) The defining points in the first (resp., third) quadrant are consecutive minimal (resp., maximal) points of the subset of P_v in that quadrant. There are $O(n_v)$ such pairs.

A point p of the third quadrant is a *maximal point* if there is no point in the third quadrant that is to the right of p and higher than p. Denote the sequence of maximal points of the third quadrant by E. A point p of the first quadrant is a *minimal point* if there is no point in the first quadrant that is to the left of p and lower than p. Denote the sequence of minimal points of the first quadrant by F. Both lists E and F are sorted from left to right (or, equivalently, from top to bottom).

Consider a consecutive pair (a, b) in E (with a to the left and above b). Let M_1 be the unique maximal P-empty rectangle whose right edge is anchored at the y-axis, its left edge passes through a, its bottom edge passes through b, and its top edge passes through some point c (in the second quadrant); it is possible that c does not exist, in which case some minor modifications (actually, simplifications) need to be applied to the forthcoming analysis, which we do not spell out.

Let M_2 be the unique maximal empty rectangle whose top edge is anchored at the x-axis, its left edge passes through a, its bottom edge passes through b, and its right edge passes through some point d (in the fourth quadrant; again, we ignore the case where d does not exist). See Figure 9. Our maximal empty rectangle cannot extend higher than c, nor can it extend to the right of d. Hence its two other defining points must be a pair (w, z) of consecutive elements of F, both lying to the left of d and below c. The minimal points which satisfy these constraints form a contiguous subsequence of F.

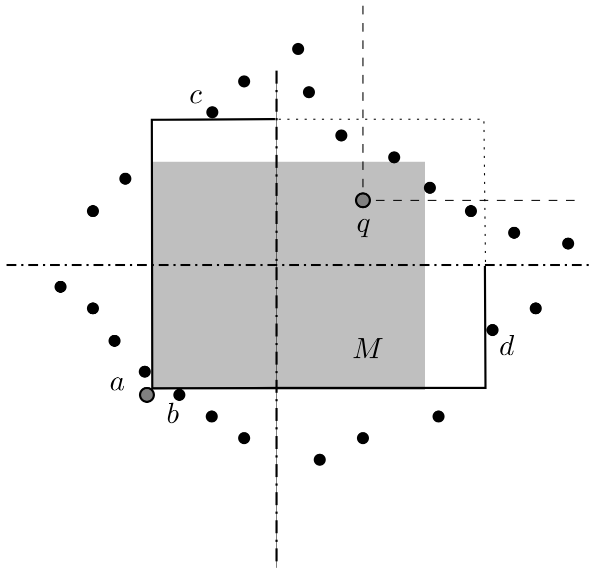

Figure 9: The structure of maximal empty rectangles with two defining points in each of the first and third quadrants.

That is, for each consecutive pair $\rho = (a, b)$ of points of E we have a contiguous "interval" $I_\rho \subseteq F$, so that any consecutive pair $\pi = (w, z)$ of points in I_ρ defines with ρ a maximal empty rectangle which contains the origin, and these are the only pairs which can define with ρ such a rectangle. (Note that we can ignore the "extreme" rectangles defined by a, b, c, and the highest point of I_ρ, or by a, b, d, and the lowest point of I_ρ, since these rectangles have three of their defining points in a common halfplane defined by the x-axis or by the y-axis, and have therefore already been treated.)

To answer queries with respect to these rectangles, we process the data as follows. We compute the chain E of maximal points in the third quadrant and the chain F of minimal points in the first quadrant. This is done in $O(n_v \log n_v)$ time by sorting the points by their y coordinate, and traversing them in order. For example, E can be constructed by scanning the points in the third quadrant from right to left maintaining the highest point seen so far; a point belongs to E if and only if it is higher than the previous highest point. For each pair $\rho = (a, b)$ of consecutive points in E we compute the corresponding delimiting points c (in the second quadrant) and d (in the fourth quadrant). Formally, c is the lowest point in the second quadrant which lies to the right of a, and d is the leftmost point in the fourth quadrant which lies above b. We then use c and d to "carve out" the interval I_ρ of F, consisting of those points that lie below c and to the left of d. We can find c by a binary search in the chain of y-minimal and x-maximal points in the second quadrant, and find d by a binary search in the chain of x-minimal and y-maximal points in the fourth quadrant. These chains can be computed in the same way as in the construction of E and F. Once we have the chains we can find, for each consecutive pair $\rho = (a, b)$ in E, the corresponding entities c, d, and I_ρ, in $O(\log n_v)$ time.

We next define a matrix A as follows. Each row of A corresponds to a pair ρ of consecutive points in E and each column of A corresponds to a pair π of consecutive points in F. If at least one point of π is not in I_ρ then the value of $A_{\rho\pi}$ is undefined. Otherwise, it is equal to the area of the (maximal empty) rectangle defined by ρ and π. By the preceding analysis, the defined entries in each row form a contiguous subsequence of columns. It is easy to verify that if ρ_2 follows (i.e., lies more to the right and below) ρ_1 on E then the left (resp., right) endpoint of I_{ρ_2} cannot be to the right of the left (resp., right) endpoint of I_{ρ_1}; See Figure 10. It follows that in each column of A the defined entries also form a contiguous subsequence of rows. Therefore, A is a partial matrix. Aggarwal and Suri [4] call this specific form of a partial matrix a *double staircase* matrix.

The following simple lemma plays a crucial role in our analysis.

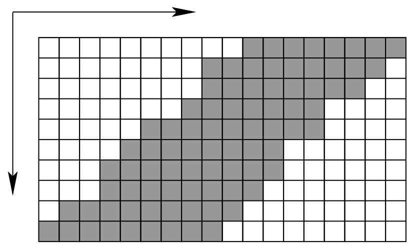
Figure 10: The structure of the defined portion of A.

LEMMA 4.1. *[33] Let x_1, x_2, y_1, y_2 be four points in the plane, so that x_1 and x_2 lie in the first quadrant, y_1 and y_2 lie in the third quadrant, x_1 lies northwest to x_2, and y_1 lies northwest to y_2. For any point w in the third quadrant and any point z in the first quadrant, let $R(w, z)$ denote the rectangle having w and z as opposite corners, and let $A(w, z)$ denote the area of $R(w, z)$. Then we have*

$$A(y_1, x_1) + A(y_2, x_2) > A(y_1, x_2) + A(y_2, x_1). \tag{4.1}$$

Proof. The situation is depicted in Figure 11. In the notation of the figure we have

$$A(y_1, x_1) + A(y_2, x_2) = A(y_1, x_2) + A(y_2, x_1) + A_1 + A_2,$$

where A_1 and A_2 are the areas of the two shaded rectangles. □

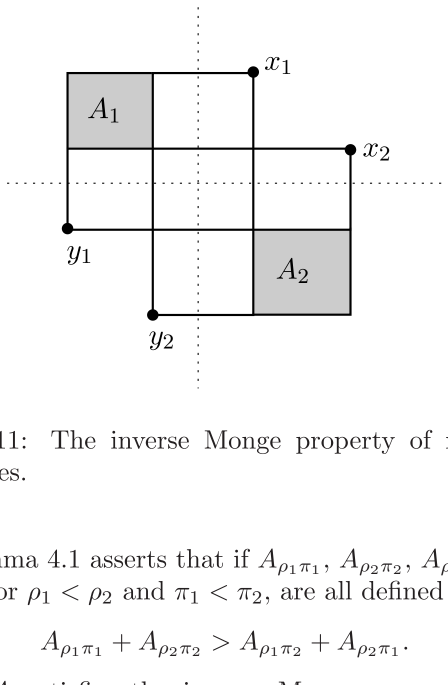

Figure 11: The inverse Monge property of maximal rectangles.

Lemma 4.1 asserts that if $A_{\rho_1\pi_1}$, $A_{\rho_2\pi_2}$, $A_{\rho_1\pi_2}$, and $A_{\rho_2\pi_2}$, for $\rho_1 < \rho_2$ and $\pi_1 < \pi_2$, are all defined then

$$A_{\rho_1\pi_1} + A_{\rho_2\pi_2} > A_{\rho_1\pi_2} + A_{\rho_2\pi_1}.$$

Hence A satisfies the inverse Monge property, with respect to its defined entries, so it is an inverse Monge partial matrix.

We construct the submatrix maximum data structure of Lemma 3.4 over A, which we use to find the maximum in A within a range of consecutive pairs in E and a range of consecutive pairs in F.

We separate the queries into two case, depending on which quadrant q lies in.

(i) Answering a query in the first (or third) quadrant. The query point q itself, if it lies in the first quadrant, defines a contiguous subsequence J_q of the sequence F of minimal points in the first quadrant, namely, those that lie above q and to its right. Only consecutive pairs with at least one point within this subsequence can form the top and right defining points of a maximal empty rectangle containing q of the type considered here. So we compute J_q, in logarithmic time, and compute the maximum in the submatrix of A defined by the set of columns of pairs overlapping J_q, using the submatrix maximum data structure, and output the corresponding rectangle.

A query with a point in the third quadrant is handled in a fully symmetric manner, using a symmetric data structure in which the roles of E and F are interchanged.

(ii) Answering a query in the second (or fourth) quadrant Consider next the case where q is in the second quadrant of B_v (the case where q is in the fourth quadrant is handled in a symmetric manner). Consider the prefix F_q of F consisting of points whose y-coordinate is larger than that of q, and the prefix E_q of E consisting of points whose x-coordinate is smaller than that of q. The rectangles defined by pairs of consecutive points in E and in F which contain q are exactly those defined by pairs with at least one point in E_q and at least one point in F_q. See Figure 12.

We find the maximum entry of A in the submatrix defined by pairs of E_q and pairs of F_q, using the submatrix maximum data structure.

Analysis: We next bound the storage, preprocessing cost, and query time for the entire structure.

For a secondary node v in T, the submatrix maximum structure requires $O(n_v\alpha(n_v)\log^2 n_v)$ time to construct and $O(n_v\alpha(n_v)\log n_v)$ space. Summing over all secondary nodes in T, we obtain that the size of the entire range tree (including the submatrix maximum data structure at each node) is $O(n\alpha(n)\log^3 n)$, and it can be constructed in $O(n\alpha(n)\log^4 n)$ time.

A query q takes $O(\log^2 n_v)$ time in each secondary node v in q's search set. (This can be reduced to $O(\log n_v)$ in case q is in the first or third quadrant of the subproblem defined by v, since in this case we can use the slab maximum data structure of Lemma 3.6). Summing over all secondary nodes v in the search set of q, yields an overall $O(\log^4 n)$ query time in the range

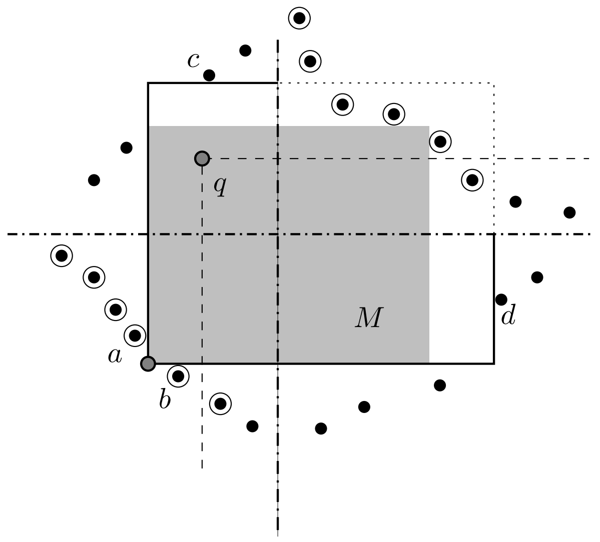

Figure 12: Querying with a point in the second quadrant. The highlighted points form the prefixes E_q and F_q, plus one extra point in each chain.

tree.

We recall that the entire presentation caters to maximal P-empty rectangles having two defining points in the first quadrant of B_v and two in the third quadrant. To handle rectangles having two defining points in each of the second and fourth quadrants, we prepare a second, symmetric version of the structure in which the roles of quadrants are appropriately interchanged, and query both structures with q.

In summary, we obtain the following main result of this section.

THEOREM 4.1. *The data structure described above requires $O(n\alpha(n)\log^3 n)$ storage, and can be constructed in $O(n\alpha(n)\log^4 n)$ time. Using the structure, one can find the largest-area P-empty rectangle contained in B and containing a query point q in $O(\log^4 n)$ time.*

Proof. Our data structure contains a segment tree S storing all maximal empty rectangles except those defined in Section 4.2. This segment tree takes $O(n \log n)$ space and is constructed in $O(n\log^4 n)$ time. A query in S takes $O(\log^2 n)$ time. The second component of our data structure is the range tree T with a submatrix maximum data structure in each secondary node. As discussed above, T takes $O(n\alpha(n)\log^3 n)$ space, $O(n\alpha(n)\log^4 n)$ preprocessing time, and answers queries in $O(\log^4 n)$ time. These bounds dominate the performance of the entire data structure. □

5 Data structure for dynamic shortest path queries with negative edge weights in planar graphs.

We next present the second application of the data structure of Section 3. Let $G = (V, E)$ be a weighted directed planar graph, where the weights of the edges of G are arbitrary real numbers, possibly negative. In this section we construct a data structure that allows to update the weight of an edge, and to query for the distance between two arbitrary nodes.

We assume that G is simple, strongly connected, and that the degree of every node of G is bounded by a constant. We also assume that G is embedded in the plane (a *plane graph*). All these assumptions can be made without loss of generality; in particular, a plane embedding of G can be found in linear time (see, e.g., [25]). In addition, we assume that there are no cycles whose total weight is negative, if such a cycle is created due to a weight update, then our data structure can detect it. We denote the number of nodes by n. Since G is planar and simple, we have $|E| = O(n)$.

By a *piece* of G we refer to a subgraph of G. An r-*division* is a division of G into $O(n/r)$ connected pieces such that each edge of G belongs to exactly one piece, and each piece contains $O(r)$ nodes. Furthermore, only $O(\sqrt{r})$ of the nodes of a piece are shared with other pieces. We call a node which lies in more than one piece a *border node*. Note that since the pieces are edge disjoint and each node has a constant degree it follows that each border node is contained in a constant number of pieces.

We can construct an r-division of an n-node planar graph in $O(n \log n)$ time [22]. We assume that each piece "inherits" its embedding from G. We define a *hole* of a piece to be a face of the piece that is not a face of G. We are interested in an r-division in which each piece has only a constant number of holes, and such an r-division can be constructed within the same $O(n \log n)$ time bound [31]. We denote by $d_P(u, v)$ the distance from a node u to a node v within the piece P.

Our data structure is based on the data structure of Klein [30] which only supports non-negative edge weights. We overcome this restriction by using reduced costs [27] which transform the problem into one with non-negative weights. Klein's data structure uses a data structure of of Fakcharoenphol and Rao [20] that implements Dijkstra's algorithm efficiently. When using reduced costs, certain range minimum data structures used by Fakcharoenphol and Rao must be reconstructed after each distance query or weight update. We replace the range minimum data structures used by Fakcharoenphol and Rao with our row-interval minimum data structure from Lemma 3.1 which has a faster construction time. We note that this idea, of using our new row-interval minimum data structure in Fakcharoenphol and Rao's fast implementation of Dijkstra's algorithm, has been recently used in the context of computing a maximum flow in planar graphs [10]. The application to data structures for reporting shortest paths is new.

5.1 Klein's dynamic algorithm. In this section we describe the data structure of Klein [30] which supports distance queries between pairs of nodes as well as updates of edge weights. The structure allows only non-negative edge weights and each query and update takes $O(n^{2/3}\log^{5/3} n)$ time. Constructing the data structure takes $O(n \log n)$ time and the space required is $O(n)$.

The structure uses the *dense distance graph* of G, which is defined with respect to an r-division of G as follows. The nodes of the graph are the border nodes of the r-division. If u and v are border nodes of a piece P then the dense distance graph contains an edge (u, v) of weight $d_P(u, v)$. It follows that the dense distance graph is the union of cliques, where each clique contains the border nodes of a single piece. The distances between all pairs of border nodes in a piece P can be computed in $O(r \log r)$ time using a multiple source shortest paths algorithm (the main result in [30]). The dense distance graph can therefore be constructed in $O(n \log n)$ time.

The algorithm for answering a distance query from a node s in a piece P to a node t in a piece P' consists of three steps:

1. In the first step we find the distance inside P from s to every border node of P, in $O(r \log r)$ time using Dijkstra's algorithm.[6] If $P = P'$, that is, s and t are in the same piece, then this step also computes $d_P(s, t)$ (which of course does not have to be equal to the shortest distance between s and t).

2. In the second step we run an implementation of Dijkstra's algorithm due to Fakcharoenphol and Rao [20] on the dense distance graph, initializing the distance labels of the border nodes of P with their distance $d_P(s, \cdot)$ from s computed in the first step, and initializing the labels for all other nodes to ∞. Fakcharoenphol and Rao's implementation of Dijkstra's algorithm runs on the dense distance graph of G in $O((n/\sqrt{r}) \log^2 r)$ time and $O(n)$ space. We describe the interface of this implementation in Section 5.2.

3. In the third step, we compute the distance from the border nodes of P' to t inside P', where the initial distance labels of the border nodes of P' are from the previous step. This also takes $O(r \log r)$ time using Dijkstra's algorithm inside P'. Upon termination of the third step we have the weight ℓ of the shortest path from s to t containing a border node. If s and t are in different pieces the distance between them is ℓ, and if they are in the same piece then the distance is $\min\{d_P(s, t), \ell\}$.

To update the weight of an edge e in a piece P, we reconstruct the part of the dense distance graph associated with P in $O(r \log r)$ time.

To conclude, distance queries to Klein's data structure take $O(r \log r + (n/\sqrt{r}) \log^2 r)$ time, and the time for a weight update of an edge is $O(r \log r)$. By setting $r = n^{2/3} \log^{2/3} n$, we get that the time for each update and query is $O(n^{2/3} \log^{5/3} n)$.

5.2 Fast Dijkstra on the dense distance graph. In this section we describe the interface of Fakcharoenphol and Rao's fast implementation of Dijkstra's algorithm [20, Sections 3.2.2 and 4].

Recall that the dense distance graph consists of a union of cliques, where each clique contains the border nodes of a single piece in the r-division. Also recall that the border nodes in each piece in the r-division are incident to a constant number of holes (faces) of that piece. We represent the dense distance graph as a set of matrices, where for each piece P and each pair of holes h and h' (possibly $h = h'$) of P there is a matrix $M^{P,h,h'}$ whose rows correspond to the border nodes of the hole h and whose columns correspond to the border nodes of the hole h', both listed in clockwise order. The matrix element $M^{P,h,h'}_{ij}$ is the distance in P from the i^{th} node of h to the j^{th} node of h'.

Fakcharoenphol and Rao's algorithm takes as input the dense distance graph H, represented by the set of matrices described above, and initial distance labels of the nodes of H. It also requires as input certain range-minimum data structures that we describe below.

The algorithm implements Dijkstra's algorithm on H starting with the given initial distance labels and takes $O(n_H \log^2 n_H)$ time, where n_H is the number of nodes of H. Note that the time dependency is on the number of nodes in H, which is $\frac{n}{\sqrt{r}}$, and not on the number of edges of H, which is $O(n)$.

The fast running time of Fakcharoenphol and Rao's algorithm is achieved by exploiting the Monge property.[7] For every piece P and every hole h of P, the

[6]Klein [30] uses a multiple source shortest path data structure for this purpose, and finds the distances in $O(\sqrt{r} \log r)$ time, but this does not affect the overall asymptotic cost of the algorithm.

[7]Some of the matrices that we describe next are Monge

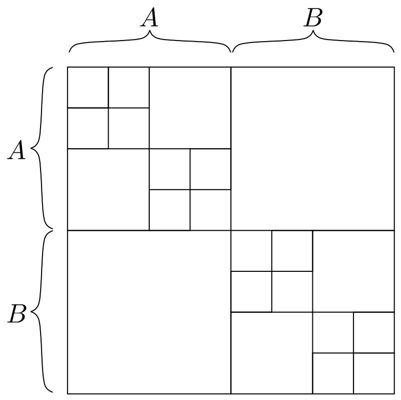

Figure 13: Illustration of the decomposotion of $M^{P,h,h}$ into Monge Submatrices.

matrix $M^{P,h,h}$ is not Monge. Fakcharoenphol and Rao split $M^{P,h,h}$ into Monge submatrices as follows. Split the border nodes of h into two consecutive subsets A and B of equal size. Since A and B are disjoint and lie on the boundary of the same face of P, the submatrix of $M^{P,h,h}$ whose rows correspond to the nodes of A and whose columns correspond to the nodes of B is a Monge matrix. Similarly, the submatrix of $M^{P,h,h}$ whose rows correspond to the nodes of B and whose columns correspond to the nodes of A is a Monge matrix. We recursively split A and B in the same way, until we get to sets of size 1. See Fig. 13 for an illustration. Together, all the submatrices defined by this recursive partitioning of the border nodes of h are disjoint and cover $M^{P,h,h}$. Note that each border node is represented in $O(\log r)$ submatrices.

For two holes $h \neq h'$ the matrix $M^{P,h,h'}$ is not Monge either. However, since in this case the rows and columns of $M^{P,h,h'}$ correspond to disjoint sets of nodes on two different holes, $M^{P,h,h'}$ can be replaced, when constructing the dense distance graph, by two Monge matrices without affecting the outcome of shortest paths computations. See [36, Section 4.4] for details. For notational convenience we regard these two Monge matrices as the two submatrices of $M^{P,h,h'}$.

We can finally describe the range-minimum data structures that are required as input by Fakcharoenphol and Rao's implementation of Dijkstra's algorithm. The algorithm requires,[8] for every piece P and every pair of holes h and h' (possibly $h = h'$), data structures capable of answering row range-minimum queries of the form $min_{s \leq \ell \leq t} M_{k,\ell}$ in $O(\log r)$ time for every possible interval $[s, t]$ of every row k of every submatrix M of $M^{P,h,h'}$. In their paper this is achieved using a separate interval tree for every row of every submatrix. As Fakcharoenphol and Rao note, the time to compute these interval trees for each matrix M is linear in the size of M. Hence the total time to construct all necessary interval trees is $\Theta(n)$.

matrices while others are inverse Monge matrices. To avoid clutter we do not make the distinction and refer to both types as Monge matrices.

[8]The assumption that these interval trees are required as input is mentioned in [20] in a footnote on page 884.

5.3 Negative edge weights. In this section we extend Klein's data structure as described in Section 5.1 and allow negative edge weights. We deal with negative edge weights using *reduced costs* [27]. Reduced costs are defined with respect to a *price function* assigning to every node v a real value (price) $\phi(v)$. The reduced cost of an edge (u, v) with weight $w(u, v)$ is $w(u, v) + \phi(u) - \phi(v)$. If the distance between two nodes u and v with respect to the original edge weights is $\ell(u, v)$, then the distance between the nodes with respect to the reduced costs is $\ell(u, v) + \phi(u) - \phi(v)$.

A price function is called *feasible* if the reduced cost of every edge is non-negative. We can get a feasible price function ϕ by choosing an arbitrary node v, and setting $\phi(u)$ to be the distance from v to u with respect to the original edge weights.

Instead of computing distances in a graph with negative edge weights, it is possible to compute distances with respect to a feasible price function using Dijkstra's algorithm. If we want to initialize the distance label of a node v with respect to original weights to some value $\delta(v)$ then with reduced costs we initialize it to $\delta(v) - \phi(v)$. After Dijkstra's algorithm finishes the (true) distance to a node u is $\delta(u) + \phi(u)$.

In our setting we use two types of price functions. First, for every piece P we have a price function ϕ_P that is used to compute distances inside P (in the first and the third steps of the query). Second, we have a price function ϕ_{dd} for the dense distance graph. Note that the prices of the same node in each of the price functions may be different. By using multiple price functions we can limit an update to a single piece and perform it efficiently.

Initially, at construction time, we choose an arbitrary node v, and compute the distances from v to all other nodes. We use these distances as the price functions ϕ_P for every piece P and as ϕ_{dd}. We can find these distances in $O(n \log^2 n / \log \log n)$ time [36]. This dominates the construction time of the data structure.

We change the query algorithm to use reduced costs rather than original weights. In the first step of the query, we run Dijkstra's algorithm inside the piece P containing s and find the distances from s to all nodes of P as described before. We run Dijkstra's algorithm with the label of s initialized to $-\phi_P(s)$ and with the reduced costs with respect to ϕ_P. This yields distance labels $\delta(v)$ for all $v \in P$. We store $\delta(v) + \phi_P(v)$ in $d_P(s, v)$, for every $v \in P$.

In the second step we compute the distance from the the border nodes of P using reduced costs with respect to $\phi_{dd}(v)$ by Fakcharoenphol and Rao's implementation of Dijkstra's algorithm. We initialize the distance label of each such border node v of P to $d_P(s, v) - \phi_{dd}(v)$. When this run of Dijkstra completes we recover, for each border node u of the piece P' containing t, the distance to u by adding $\phi_{dd}(u)$ to its computed distance label. We similarly run the third step using reduced costs with respect to $\phi_{P'}$.

There are two obstacles in calling Fakcharoenphol and Rao's implementation of Dijkstra's algorithm on the dense distance graph using reduced costs in the second step. The first obstacle is that we cannot afford to explicitly compute the reduced costs of the edges of the dense distance graph since that would take $O(n)$ time. Instead, whenever the implementation requires the reduced cost of an edge (u, v) in the dense distance graph, we compute it on the fly by adding $\phi_{dd}(u)$ to $d_P(u, v)$ and subtracting $\phi_{dd}(v)$.

The second obstacle is that we need to recompute the range minimum data structures required by Fakcharoenphol and Rao's implementation of Dijkstra's algorithm. These data structures now have to answer range minimum queries with respect to reduced costs rather than original weights, and therefore we have to update them when the potential function ϕ_{dd} changes. As we noted in Section 5.2, computing all necessary interval trees used by Fakcharoenphol and Rao would take $\Theta(n)$ time, which we cannot afford. Instead, we replace the per-row range minimum structures that Fakcharoenphol and Rao use with a single row-interval minimum data structure of Lemma 3.1 for each Monge submatrix M. The overall construction time of these data structures is faster, so we can efficiently support the change in the price function.[9]

We bound the time it takes to compute all these data structures by considering the Monge submatrices of each matrix $M^{P,h,h'}$ separately. Computing the row-interval minimum data structure of Lemma 3.1 for a matrix with x rows and columns takes $O(x \log x)$ time. For a fixed P and h, the matrix $M^{P,h,h}$ has $O(\sqrt{r})$ rows and columns, so the time required for constructing the data structure all $O(\log r)$ Monge submatrices of $M^{P,h,h}$ is $O(\sqrt{r} \log^2 r)$. Similarly for fixed P and $h \neq h'$, the time required for the two Monge submatrices of $M^{P,h,h'}$ is $O(\sqrt{r} \log r)$. Since the number of holes in each piece is constant, the total time for all matrices of a piece P is $O(\sqrt{r} \log^2 r)$. Summing over all $\frac{n}{r}$ pieces we get that the total construction time for all of the row-interval data structures is $O(\frac{n}{\sqrt{r}} \log^2 r)$ which is the same as the time required by a single call to Fakcharoenphol and Rao's implementation of Dijkstra's algorithm. Note that the total space required for our data structure remains linear.

When we initialize the data structure and construct the dense distance graph, we compute these row-interval minimum data structures with respect to the initial potential function defined above.

Now consider an update of the weight of an edge $e = (u, v)$ in a piece P to a new value w. Before updating the weight of e, we compute the distance from v to u, using the distance query algorithm. If this distance is less than $-w$, then changing the weight of e to w will create a negative cycle. Otherwise observe that the current distances from v can be used as a feasible price functions both before and after the weight of e is updated, since a shortest path from v to any node cannot use e. We compute the distances from v in P as in the first step of the query algorithm and use these distances as the new price function ϕ_P for the piece P. Next, we compute distances from v to every border node as in the second step of the query algorithm, and use these distances as the new price function ϕ_{dd} for the dense distance graph. The time it takes to find the distances from v to all nodes in P and to all border nodes is $O(r \log r + (n/\sqrt{r}) \log^2 r)$, since this is the time for the distance query. The change in the weight of e may change the distances inside P. We therefore recompute the distances between all border nodes of P inside P in $O(r \log r)$ time and update the distance matrices of the piece P, as in the data structure for non-negative edge-weights described in Section 5.1 (here as well we use the distances from v in P as a feasible price function since Klein's multiple source shortest paths algorithms requires non-negative edge weights). Since we have updated the price function of the dense distance graph we must recompute all row-interval minimum data structures. This takes $O((n/\sqrt{r}) \log^2 r)$ time as argued above.

In conclusion, we obtained the following theorem:

THEOREM 5.1. *There is a data structure that, given a weighted directed planar graph G with n vertices, supports distance queries between pair of nodes and updates to edge weights in $O(n^{2/3} \log^{5/3} n)$ time per operation. The data structure can be constructed in $O(n \log^2 n / \log \log n)$ time and requires $O(n)$ space.*

[9] A Monge matrix remains Monge when we add a constant to a row or a column. Therefore each submatrix of M remains Monge when we think of the entry corresponding to border nodes u and v as containing $d_P(u, v) + \phi(u) - \phi(v)$ rather than $d_P(u, v)$.

Proof. The correctness of the data structure follows from the description in this section by induction on the number of operations. By the analysis above the construction time is as claimed, and the running time for a query or an update is $O(r \log r + (n/\sqrt{r}) \log^2 r)$. Choosing $r = n^{2/3} \log^{2/3} n$ yeilds the claimed time bounds. □

References

[1] Aggarwal, A., and Klawe, M. Applications of generalized matrix searching to geometric algorithms. *Discret. Appl. Math. 27* (1990), 3–23.

[2] Aggarwal, A., Klawe, M., Moran, S., Shor, P., and Wilber, R. Geometric applications of a matrix-searching algorithm. *Algorithmica 2* (1987), 195–208.

[3] Aggarwal, A., and Park, J. Notes on searching in multidimensional monotone arrays. In *Proc. 29th Annu. Symp. Found. Comput. Sci.* (1988), IEEE Computer Society, pp. 497–512.

[4] Aggarwal, A., and Suri, S. Fast algorithms for computing the largest empty rectangle. In *Proc. 3rd Annu. Symp. Comput. Geom.* (1987), ACM, pp. 278–290.

[5] Atallah, M. J., and Frederickson, G. N. A note on finding a maximum empty rectangle. *Discret. Appl. Math. 13* (1986), 87–91.

[6] Augustine, J., Das, S., Maheshwari, A., Nandy, S. C., Roy, S., and Sarvattomananda, S. Querying for the largest empty geometric object in a desired location. *CoRR abs/1004.0558* (2010).

[7] Baird, H. S., Jones, S. E., and Fortune, S. J. Image segmentation by shape-directed covers. In *Proc. 10th Int. Conf. on Pattern Recognit.* (1990), vol. I, IEEE Computer Society, pp. 820–825.

[8] Bender, M. A., and Farach-Colton, M. The LCA problem revisited. In *Proc. 4th Lat. Am. Symp. Theor. Inform.* (2000), G. H. Gonnet, D. Panario, and A. Viola, Eds., vol. 1776 of *LNCS*, Springer-Verlag, pp. 88–94.

[9] Boland, R. P., and Urrutia, J. Finding the largest axis-aligned rectangle in a polygon in $O(n \log n)$ time. In *Proc. 13th Can. Conf. Comput. Geom.* (2001), pp. 41–44.

[10] Borradaile, G., Klein, P. N., Mozes, S., Nussbaum, Y., and Wulff-Nilsen, C. Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. In *Proc. 52nd Annu. Symp. on Found. of Comput. Sci.* (2011). To Appear.

[11] Burkard, R. E., Klinz, B., and Rudolf, R. Perspectives of monge properties in optimization. *Discret. Appl. Math. 70* (1996), 95–161.

[12] Cabello, S. Many distances in planar graphs. *Algorithmica*, 1–21. To appear.

[13] Chaudhuri, J., Nandy, S. C., and Das, S. Largest empty rectangle among a point set. *J. Algorithms 46* (2003), 54–78.

[14] Chazelle, B., Drysdale III, R. L., and Lee, D. T. Computing the largest empty rectangle. *SIAM J. Comput. 15* (1986), 300–315.

[15] Chazelle, B., and Guibas, L. J. Fractional cascading: I. A data structuring technique. *Algorithmica 1* (1986), 133–162.

[16] Chen, D. Z., and Xu, J. Shortest path queries in planar graphs. In *Proc. 32nd Annu. ACM Symp. Theory Comput.* (2000), ACM, pp. 469–478.

[17] Chew, L. P., and Dyrsdale III, R. L. Voronoi diagrams based on convex distance functions. In *Proc. 1st Annu. Symp. Comput. Geom.* (1985), ACM, pp. 235–244.

[18] de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer-Verlag, Berlin, 2008.

[19] Djidjev, H. Efficient algorithms for shortest path queries in planar digraphs. In *Proc. 22nd Int. Workshop Graph Theor. Concept Comput. Sci.* (1997), F. d'Amore, P. Franciosa, and A. Marchetti-Spaccamela, Eds., vol. 1197 of *LNCS*, Springer-Verlag, pp. 151–165.

[20] Fakcharoenphol, J., and Rao, S. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci. 72* (2006), 868–889.

[21] Feuerstein, E., and Marchetti-Spaccamela, A. Dynamic algorithms for shortest paths in planar graphs. *Theor. Comput. Sci. 116* (1993), 359–371.

[22] Frederickson, G. N. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput. 16* (1987), 1004–1022.

[23] Hershberger, J. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Inf. Process. Lett. 33* (1989), 169–174.

[24] Hoffman, A. J. On simple linear programming problems. In *Proc. Symp. Pure Math.*, vol. VII. Amer. Math. Soc., 1963, pp. 317–327.

[25] Hopcroft, J., and Tarjan, R. Efficient planarity testing. *J. ACM 21* (1974), 549–568.

[26] Italiano, G. F., Nussbaum, Y., Sankowski, P., and Wulff-Nilsen, C. Improved algorithms for min cut and max flow in undirected planar graphs. In *Proc. 43rd Annu. ACM Symp. Theory Comput.* (2011), ACM, pp. 313–322.

[27] Johnson, D. B. Efficient algorithms for shortest paths in sparse networks. *J. ACM 24* (1977), 1–13.

[28] Klawe, M. M. Superlinear bounds for matrix searching problems. *J. Algorithms 13* (1992), 55–78.

[29] Klawe, M. M., and Kleitman, D. J. An almost linear time algorithm for generalized matrix searching. *SIAM J. Discret. Math. 3* (1990), 81–97.

[30] Klein, P. N. Multiple-source shortest paths in planar graphs. In *Proc. 16th Annu. ACM-SIAM Symp. on Discret. Algorithm.* (2005), SIAM, pp. 146–155.

[31] Klein, P. N., and Subramanian, S. A fully dynamic approximation scheme for shortest paths in planar graphs. *Algorithmica 22* (1998), 135–249.

[32] Leven, D., and Sharir, M. Planning a purely translational motion for a convex object in two-dimensional space using generalized voronoi diagrams. *Discret. Comput. Geom. 2* (1987), 9–31.

[33] McKenna, M., O'Rourke, J., and Suri, S. Finding the largest rectangle in an orthogonal polygon. In *Proc. 23rd Allerton Conf. Commun. Control Comput.* (1985), pp. 486–495.

[34] Monge, G. Mémoire sur la théorie des déblais et des remblais. In *Histoire de l'Académie Royale des Science.* 1781, pp. 666–704.

[35] Mozes, S., and Sommer, C. Exact distance oracles for planar graphs. In *Proc. 23rd Annu. ACM-SIAM Symp. on Discret. Algorithm.* (2012). To appear.

[36] Mozes, S., and Wulff-Nilsen, C. Shortest paths in planar graphs with real lengths in $O(n \log^2 n / \log \log n)$ time. In *Proc. 18th Eur. Symp. Algorithm. (2)* (2010), M. de Berg and U. Meyer, Eds., vol. 6347 of *LNCS*, Springer-Verlag, pp. 206–217.

[37] Naamad, A., Lee, D. T., and Hsu, W. L. On the maximum empty rectangle problem. *Discret. Appl. Math. 8*, 3 (1984), 267 – 277.

[38] Nandy, S. C., Sinha, A., and Bhattacharya, B. B. Location of the largest empty rectangle among arbitrary obstacles. In *Proc. 14th Conf. Found. Softw. Technol. Theor. Comput. Sci.* (1994), P. S. Thiagarajan, Ed., vol. 880 of *LNCS*, Springer-Verlag, pp. 159–170.

[39] Nussbaum, Y. Improved distance queries in planar graphs. In *Proc. 11th Int. Symp. Algorithm. Data Struct.* (2011), F. Dehne, J.-R. Sack, and R. Tamassia, Eds., vol. 6844 of *LNCS*, Springer-Verlag, pp. 642–653.

[40] Park, J. K. A special case of the n-vertex traveling-salesman problem that can be solved in $O(n)$ time. *Inf. Process. Lett. 40*, 5 (1991), 247 – 254.

[41] Sharir, M., and Agarwal, P. K. *Davenport-Schinzel Sequences and their Geometric Applications.* Cambridge University Press, New York, NY, USA, 1995.

[42] Ullman, J. D. *Computational Aspects of VLSI.* W. H. Freeman & Co., New York, NY, USA, 1984.