Submatrix maximum queries in Monge matrices and partial Monge 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 partial Monge matrices, where a query seeks the maximum element in a contiguous submatrix of the given matrix. The structure, for an $n \times n$ Monge matrix, takes $O(n \log n)$ space, $O(n \log^2 n)$ preprocessing time, and answers queries in $O(\log^2 n)$ time. For partial Monge matrices the space and preprocessing grow by $\alpha(n)$ (the inverse Ackermann function), and the query remains $O(\log^2 n)$. Our design exploits an interpretation of the column maxima in a Monge (resp., partial Monge) matrix as an upper envelope of pseudo-lines (resp., pseudo-segments).

We give two applications: (1) For a planar set of n points in an axis-parallel rectangle B, we build a data structure, in $O(n\alpha(n) \log^4 n)$ time and $O(n\alpha(n) \log^3 n)$ space, that returns, for a query point p, the largest-area empty axis-parallel rectangle contained in B and containing p, in $O(\log^4 n)$ time. This improves substantially the nearly-quadratic storage and preprocesing obtained by Augustine et al. [arXiv:1004.0558]. (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 graph-distance queries between arbitrary pairs of nodes in $O(n^{2/3} \log^{5/3} n)$ time per operation. This improves a previous algorithm 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].

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] A typical situation where Monge matrices arise is the following. Suppose we have two vertically ordered sets of points,

*An extended abstract of this paper was presented at [34].

†School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E-mail: haimk@post.tau.ac.il, yahav.nussbaum@cs.tau.ac.il. Work by Haim Kaplan and Yahav Nussbaum were partially supported by Grant 2006/204 from the U.S.–Israel Binational Science Foundation, by Grant 822/10 from the Israel Science Fund, and by the Israeli Centers of Research Excellence (I-CORE) program.

‡Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139, USA. E-mail: shaym@mit.edu. Part of the work by Shay Mozes was conducted while at Brown University, and partially supported by NSF Grants CCF-0964037, CCF-1111109 and by a Kanellakis fellowship.

§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. 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.

[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.

A and B, on the left and right vertical sides of an axis-parallel rectangle, respectively. Monge [43] observed that, for $i < j$ and $k < \ell$ any path in the rectangle 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 from the triangle inequality 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 [51]. The greedy algorithm solves the trasportation problem optimally if the costs form a Monge matrix [30]. Monge matrices were also used to obtain efficient algorithms for several problems on convex n-gons, like finding the k furthest vertices from any vertex [2], finding a minimum-area circumscribing convex d-gon, and others [2, 3]. For a survey on Monge matrices and their uses in combinatorial optimization see [12].

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 *partial* Monge matrices in which some of the entries are undefined [1, 36, 37]. In these cases we require that the matrix have the Monge property only with respect to the defined entries. 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.

For the special case in which 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, $O(n \log n)$ space, and can answer queries in $O(\log n)$ time. Note that when the query submatrix is a *slab*, that is, a contiguous subsequence of complete rows, it is easy to obtain a data structure that can be constructed in $O(n)$ time and can answer a query in $O(1)$ time. We can achieve that by computing row maxima using the SMAWK algorithm, and then constructing an interval-maxima data structure on the array of row-maxima. The range-maxima problem on an array is well studied and several data structures with linear size, linear preprocessing time, and constant query time are known ([28], see also [8] and the references therein).

We show how to extend our data structures to *partial* Monge matrices, which are Monge matrices that contain undefined entries, so that the defined entries form within each row and column a contiguous interval (see, e.g., Figure 2). 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 $\alpha(n) \log n$, and is now $O(n\alpha(n) \log^2 n)$.

We obtain these data structures using techniques from computational geometry. Specifically, we think of the rows of the matrix as tabulated functions over the discrete sequence of

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

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 maximum data structure has already been applied in a recent maximum flow algorithm for planar graphs of Borradaile et al. [10]. 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 [24] 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, for example, in databases, for queries trying to identify ranges of values that never appear together [27].

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, among other applications, in the enumeration of "maximal white rectangles" in image segmentation [7].

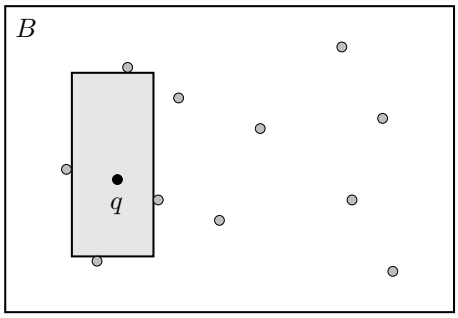


Figure 1: A maximal P-empty rectangle containing a query point q.

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 be constructed in $O(n\alpha(n) \log^4 n)$ time. Our query time is $O(\log^4 n)$, slightly worse than that of Augustine et al.

In a nutshell, our algorithm computes all the maximal P-empty rectangles and preprocesses them into a 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 8), so we cannot afford to compute them explicitly. Instead, we exploit the inverse partial Monge matrix structure that exists in certain configurations of points (so-called "double staircases"), 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. [41] and used by Aggarwal and Suri [4] for finding the (global) P-empty rectangle of largest area and the (global) P-empty rectangle of largest perimeter.

Our data structure can also be modified to find the largest-perimeter P-empty rectangle containing a given query point q.

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. The bottleneck step in our construction is a single source shortest path computation (when edges can have negative weights) [45]. We can also report the shortest path Q itself in additional $O(|Q|\log\log\Delta)$ time, where Δ is the maximum degree of a node in the path Q.

Our data structure is based on the data structure of Fakcharoenphol and Rao [24], 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 [38], which has the same query time and space bound as our algorithm but does not support negative edge weights. For the convenience of the reader, we provide a review of these techniques before adapting and extending them to our setup.

Italiano et al. [32] 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, as well as the aforementioned bounds on storage and preprocessing.

1.2 Related work

As we mentioned the range-maxima problem on an array is well studied and several data structures with linear size, linear preprocessing time, and constant query time were developed for it. The range-maxima problem on general multidimensional arrays has also been studied, and the best data structure of Yuan and Atallah [53] is linear in the size of the matrix (i.e. $O(n^2)$), can be constructed in $O(n^2)$ time, and answers a query in constant time. Brodal et al. [11] proved that with n^2/c additional bits a query must take $\Omega(c)$ time. In contrast, our results show that on matrices with the Monge property the range minima problem is substantially easier.

Largest empty rectangles. 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 [19, 39]. There have been several studies on finding the largest-area maximal empty rectangle [5, 16, 46, 21, 49] 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" monotone chains of points, and the inverse Monge property which

they satisfy. Our main contribution is in adapting these ideas to construct a *data structure* for finding the largest empty rectangle containing a query point.

Nandy et al. [47] show how to find the largest-area axis-parallel empty rectangle avoiding a set of polygonal obstacles in $O(n \log^2 n)$ time and $O(n)$ space. 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. [15] 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] also 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.[3]

Distance queries in planar graphs. Static data structures that preprocess an input planar graph to answer distance queries efficiently were studied by many authors [13, 18, 22, 24, 25, 26, 38, 44, 48]. The dynamic setting in which updates of edge weights are supported was recently studied by Fakcharoenphol and Rao [24]. 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 that supports negative edge weights but with query and update time of $O(n^{4/5} \log^{13/5} n)$, with the same space and preprocessing time. Klein [38], using the 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 $k < \ell$, $M_{ik} < M_{i\ell}$ implies $M_{jk} < M_{j\ell}$. 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, in linear time, 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, but the defined entries of each row are continuous, and the defined entries of each column are

[3] In [35], Kaplan and Sharir have recently improved the storage and preprocessing costs to nearly linear in n.

continuous.[4] See Figure 2. A partial totally monotone (resp., Monge, inverse Monge) matrix is a partial matrix whose defined entries satisfy the total monotonicity (resp., Monge, inverse Monge) condition. More precisely, the respective condition has to hold for every quadruple of entries, M_{ik}, $M_{i\ell}$, M_{jk}, $M_{j\ell}$, all of which are defined. (Note that in this case the entire submatrix with these four entries as "corners" is defined.)

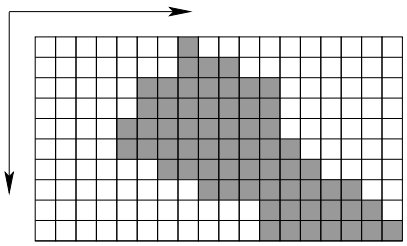


Figure 2: The defined part of a partial matrix.

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 totally defined function $\ell(x), x \in \mathbb{R}$, and of a pseudo-segment s as a partially defined function $s(x), x \in I_s$, where $I_s \subseteq \mathbb{R}$ is some (possibly unbounded) interval. The *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 $\mathcal{E}_S(x) = \max\{s(x) \mid s \in S, x \in I_s\}$ 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 $\mathcal{E}_L$. A *breakpoint* in the upper envelope $\mathcal{E}_S$ of a set S of pseudo-segments is either an intersection point of two pseudo-segments on $\mathcal{E}_S$ or an endpoint of one of the pseudo-segments on $\mathcal{E}_S$. We define the *complexity* of an envelope $\mathcal{E}_L$ or $\mathcal{E}_S$ 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 $\mathcal{E}_L$ is $O(|L|)$ (it is in fact at most $|L|-1$). The complexity of $\mathcal{E}_S$ is known to be $O(|S|\alpha(|S|))$ [52].

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 inverse Monge property of M we get that $M_{ik} - M_{jk} \geq M_{i\ell} - M_{j\ell}$, for any four indices $i < j$ and $k < \ell$. In other words, if we extend the domain of definition of each function $\hat{M}_\rho$ to the real interval $[1, n]$, by linearly interpolating between each pair of π-consecutive points, then the function $\hat{M}_i(\cdot) - \hat{M}_j(\cdot)$ is weakly decreasing. It follows that if the graphs of $\hat{M}_i$ and $\hat{M}_j$ meet then they either meet at a single point, or they overlap at some interval. Assuming, for simplicity of presentation, that only the former intersection pattern can arise (which will be the case, e.g., when all the inverse Monge inequalities are sharp), then the resulting piecewise linear functions $\hat{M}_\rho$ behave

[4] 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 any partial matrix by the definition of this paper can be decomposed into two partial matrices of [1].

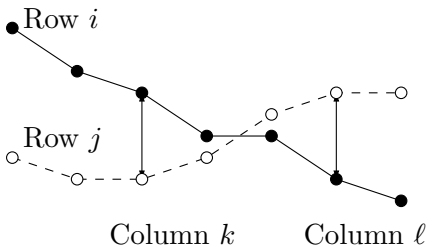


Figure 3: The pseudo-lines of two rows $i < j$ in an inverse Monge matrix.

as pseudo-lines.[5] Moreover, for $i < j$, 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 "slopes" of the pseudo-lines of the rows are arranged in increasing order. See Figure 3. Note that the same analysis applies to Monge matrices, except that the order of the "slopes" of the resulting pseudo-lines is decreasing.

Similarly if M is an $m \times n$ partial inverse Monge 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 being interpolated in the same manner as above, is defined over some connected subinterval of $[1, n]$. By the inverse Monge property we get that $M_{ik} - M_{jk} \geq M_{i\ell} - M_{j\ell}$, when $i < j$, $k < \ell$, 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.

3 The data structure

In this section we develop the main result of our paper, namely, data structures for submatrix maximum queries in (inverse) Monge matrices and in partial (inverse) Monge 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 explicitly—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 (inverse) 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 $\mathcal{E}$ 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 $\mathcal{E}$ contains only $O(m)$ breakpoints, and we use these breakpoints to implicitly represent $\mathcal{E}$.[6] 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.

The breakpoints of $\mathcal{E}$ partition the domain $[1, n]$ into *intervals*, so that $\mathcal{E}$ is attained by a single pseudo-line (i.e., row) over each interval. Every column π is contained in one of these intervals, and we can find that interval using binary search on the breakpoints. Once we found

[5] In fact it suffices that M^t is totally monotone for the functions $\hat{M}_\rho$ to behave as pseudo-lines.

[6] 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 all that really counts is the pair of consecutive columns between which the breakpoint appears. We will not refer to this issue explicitly in what follows.

the interval, we know which pseudo-line attains $\mathcal{E}$ at π (that is, which row contains the maximum of column π), and can 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 $\mathcal{E}$ as a list of its breakpoints, and of the the pseudo-line (row) that attains $\mathcal{E}$ between each pair of consecutive breakpoints.

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. (This is similar to the hierarchical representation of an upper envelope used by Overmars and van-Leeuwen to support insertions and deletions of lines [50].)

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^t 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 some row of w_1 and that of some row of w_2, and ends with a suffix of the upper envelope of w_2. We merge the sequences of breakpoints of the envelopes of w_1 and of w_2, checking at each breakpoint, in $O(1)$ time, for the relative order of the envelopes over it, and keeping the breakpoint only if it remains on the upper envelope of u. This allows us to find the interval containing the unique intersection of the envelopes. Over this interval there are only two rows, one from each subtree, "competing" for the maximum, and we find the intersection point of their pseudo-lines in $O(\log n)$ time. Then we form the new envelope of u as prescribed above, by concatenating the prefix of the upper envelope of w_1, the new breakpoint, and the suffix of the upper envelope of w_2. All these steps take $O(k + \log n)$ time. Summing these costs over all nodes of T_h, we get an overall cost of $O(m(\log m + \log n))$. The total size of T_h is $O(m \log m)$.[7]

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 [17]. 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 (asymptotically) any space or preprocessing time overhead,

[7] We could, in fact, construct T_h in $O(m \log n)$ time and represent it using only $O(m)$ space at the cost of somewhat complicating the algorithm. For that we have to do a binary search to locate the last breakpoint of the envelope of w_1 which is also on the envelope of u and the first breakpoint of the envelope of w_2 which is on the envelope of u. To avoid copying parts of the envelopes of w_1 and w_2 in order to produce the envelope of u we represent envelopes using persistent search trees and produce the envelope of u by splitting and concatenating nondestructively the search trees representing the envelopes of w_1 and w_2 in $O(\log k)$ time and extra space [23]. We do not know how to efficiently combine these persistent search trees with fractional cascading which we use to speed up the query.

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 symmetric 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 $\mathcal{B}$; A query in $\mathcal{B}$ takes $O(\log n)$ time. For every node u 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, but each such interval may appear in the upper envelope of more than one node of T_h. In fact, if we recall the construction of T_h then we can easily verify that there are only $O(m)$ distinct such intervals in all the upper envelopes of the nodes of T_h together. The reason for that is that when we merge the upper envelopes of the children w_1 and w_2 of a node u then at most two new intervals are created in the upper envelope of u. All other intervals in the upper envelope of u appear also either in the upper envelope of w_1 or in the upper envelope of w_2.

It follows that we can find the maxima in each of these intervals, when it is created, in a total of $O(m \log n)$ time. 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 set $\mathcal{I}$ of consecutive intervals of the upper envelope of u that are fully contained in C. Then we find the maximum in the range covered by the union of the intervals in $\mathcal{I}$ 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 a specific row ρ. Similarly the maximum of the upper envelope of u in the suffix s of C not covered by $\mathcal{I}$ is attained at a specific 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. Repeating this step for each of the $O(\log m)$ subtrees of T_h that comprise R, we cover the entire query submatrix. The cost of finding the maximum in the upper envelope over C of any single node u of T_h is $O(\log m + \log n)$, so the overall query time is $O(\log m(\log m + \log n))$. We thus obtain the following *submatrix maximum* data structure:

Theorem 3.2. *Given an inverse Monge matrix[8] of size $m \times n$, one can construct in $O((m +$*

[8]This theorem in fact holds for any totally monotone matrix whose transpose is also totally monotone.

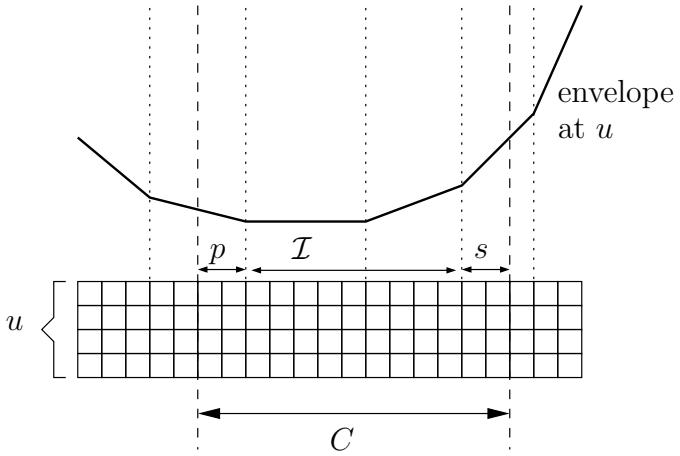


Figure 4: Maximum query at a single node u of T_h.

$n)(\log m+\log n))$ *time a data structure of size* $O(m \log m+n \log n)$ *that can report the maximum entry in any query 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 results also apply to Monge matrices.

3.2 Submatrix maximum in partial Monge matrices

We extend the two data structures from the previous section to partial inverse Monge matrices. Let M denote an $m \times n$ partial inverse Monge 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 $O(k\alpha(k))$ [52], 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, keeping only those that appear on the envelope of u, each new interval can contain a new breakpoint of the upper envelope at u. Intervals of this kind are characterized by the property that the envelope of u is attained at their left (resp., right) endpoint by the envelope of the left (resp., right) child of u. We find the new breakpoint inside each such interval in $O(\log n)$ time, and complete the construction of the envelope at u in a similar manner to the one described in the previous section. Since we pay $O(\log n)$ time to find a new breakpoint, and there are $O(m\alpha(m) \log m)$ breakpoints in total (over all nodes of the tree), the construction takes $O(m\alpha(m) \log m \log n)$ time, and the required storage 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 [29], 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 a partial totally monotone 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-interval 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 of the nodes of T_h 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 $O(m\alpha(m)\log m + n\alpha(n)\log n)$. 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:

Theorem 3.4. *Given a partial inverse Monge matrix[9] one can construct, in $O((m\alpha(m) + n\alpha(n))\log m \log n)$ time, a data structure of size $O(m\alpha(m)\log m + n\alpha(n)\log n)$ 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 partial Monge 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

As mentioned in Section 1.1, 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.

Theorem 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 partial totally monotone matrices we can construct a similar data structure using the algorithm of Klawe and Kleitman [37] for finding row maxima instead of the SMAWK algorithm.[10] This yields:

Theorem 3.6. *Given a partial totally monotone 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, which is an algorithm that preprocesses P

[9] As Theorem 3.2, this theorem also holds for any partial totally monotone matrix whose transpose is also totally monotone.

[10] The definition of partial matrices in [37] is the one in [1], which is different than the definition used in this paper. However, the algorithm of [37] does apply in our case since, as previously noted, we can decompose a partial matrix into two partial matrices in the sense of [37] and run the algorithm of [37] on each of the two matrices.

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 refer the reader to the introduction for the notation that we shall use here.

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. This data structure can store a set $\mathcal{M}$ of N axis-parallel rectangles in the plane, such that we can efficiently report all rectangles, or just find the largest rectangle, containing a query point. 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. In our case these N rectangles will be maximal P-empty rectangles, as defined in the introduction; they are therefore defined by n points so their x-projections and their y-projections have only n distinct endpoints. We first construct a standard segment tree S [20] on the x-projections of the rectangles in $\mathcal{M}$. This is a balanced binary search tree whose leaves correspond to the $O(n)$ intervals between consecutive 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. Every rectangle $R \in \mathcal{M}$ is stored 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 $\mathcal{M}_u$ in the same manner. 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 containing 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 $O(N \log^2 n)$.

A simple-minded solution would just take $\mathcal{M}$ to be the set of all maximal P-empty rectangles. Unfortunately, this may not be efficient since the size N of $\mathcal{M}$ could be quadratic in n in the worst case. (See Figure 8 for an illustration.) To get an efficient solution we will store only a subset of the rectangles of cardinality nearly linear in n, in a two-dimensional segment tree, and for the other rectangles we will use 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 a partial inverse Monge matrix, and we will use our submatrix maximum query data structure on the resulting matrix to find the largest-area P-empty rectangle containing any query point q.

4.1 Maximal rectangles with edges on the boundary of B

Let e_t, e_b, e_ℓ, and e_r be the top, bottom, left, and right edges of B, respectively. Naamad et al. [46] classified the maximal P-empty rectangles within B according to the number of their edges that touch the edges of B. They show that there are only $O(n)$ maximal P-empty rectangles with at least one edge on ∂B. 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, by searching with q in S. (The segment tree S will also store additional rectangles that will arise in later steps of the construction, but the overall number of these rectangles will still be nearly linear; see below for details.)

For the sake of completeness, we provide the easy details concerning the number of these anchored maximal P-empty rectangles and their construction cost.

(i) Three edges of R lie on ∂B. It is easy to verify that there are only four such rectangles, one for each triple of edges of B.

(ii) Two adjacent edges of R lie on ∂B. Suppose, without loss of generality, that the top and right edges of R lie on e_t and e_r, respectively. The other two edges of R must be supported by a pair of *maxima* of P (that is, points $p \in P$ for which no other point $q \in P$ satisfies $x_q > x_p$ and $y_q > y_p$), consecutive in the sorted order of the maxima (by their x- or y-coordinates). Since there are $O(n)$ pairs of consecutive maxima, the number of anchored rectangles of this kind is also $O(n)$. See Figure 5. The other three situations are handled in a fully symmetric manner.

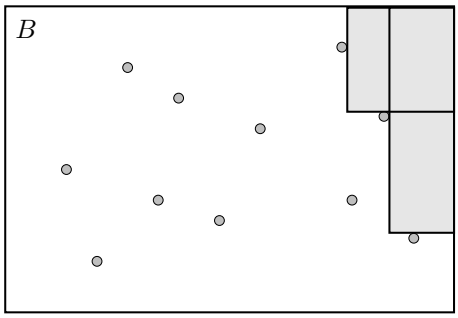


Figure 5: Maximal P-empty rectangles with two adjacent edges on ∂B.

(iii) Two opposite edges of R lie on two opposite edges of B. Suppose, without loss of generality, that the left and right edges of R lie on e_ℓ and e_r, respectively. In this case the top and bottom edges of R must be supported by two points of P, consecutive in their y-order. Clearly, there are $O(n)$ such pairs, and thus also $O(n)$ rectangles of this kind. Again, handling the top and bottom edges of B is done in a fully symmetric manner. See Figure 6.

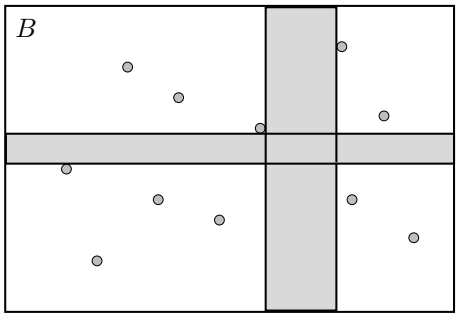


Figure 6: Maximal P-empty rectangles with two opposite edges on ∂B.

(iv) One edge of R lies on ∂B. Suppose, without loss of generality, that the right edge of R lies on e_r. Then the three other sides of R must be supported by points of P. For each point $p \in P$ there is a unique maximal P-empty rectangle whose right edge lies on e_r and whose left edge passes through p. This rectangle is obtained by connecting p to e_r by a horizontal segment h and then by translating h upwards and downwards until it first hits two respective points of P, or reaches ∂B. (In the latter situations we obtain rectangles of the preceding types.) Hence there are $O(n)$ rectangles of this kind too. See Figure 7.

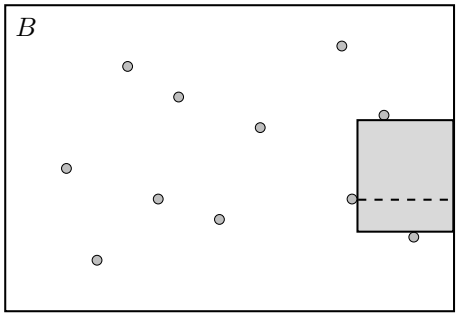


Figure 7: A maximal P-empty rectangle with one edge on ∂B.

It is easy to compute all the maximal P-empty anchored rectangles of the above four classes, in overall $O(n \log n)$ time: Computing rectangles of type (i) and (iii) only requires sorting the points by their x or y coordinates. Computing rectangles of type (ii) (of the specific kind depicted in Figure 5) requires computing the list of maximal points. This can be done by scanning the points from right to left maintaining the highest point seen so far. A point p is maximal if and only if it is higher than the previous highest point. We can compute the rectangles of type (iv) (of the specific kind depicted in Figure 7) also by traversing the points from right to left while maintaining the points already traversed, sorted by their y-coordinates, in a balanced search tree. When we process a point p then its successor and predecessor (if they exist) in the tree define the top and bottom edges of the rectangle of type (iv) whose left edge passes through p.

We collect these rectangles and store them in our two-dimensional segment tree S. Given a query point q, we can find the rectangle of largest area containing q among these rectangles by searching in S, as explained above, in $O(\log^2 n)$ time.

4.2 Maximal empty rectangles supported by four points of P

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 [46]; see Figure 8 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.

We store the points of P in a two-dimensional range tree (see, e.g., [20]). 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

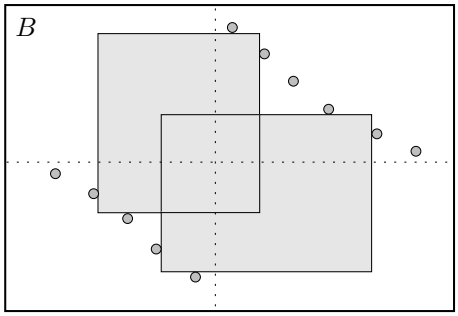


Figure 8: A set P of n points with $\Theta(n^2)$ maximal P-empty rectangles.

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 internal 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 9.

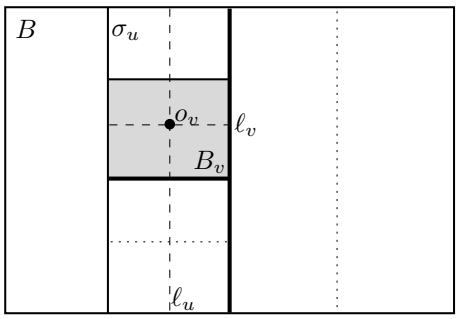


Figure 9: The rectangle B_v associated with a secondary node v, with its splitters and origin.

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 (or rather inherits from u) 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 10. Note that both v and u are internal nodes (each being a lowest common ancestor of two distinct leaves) so o_v is indeed defined. Furthermore, one can easily verify that v is in the search set of q.

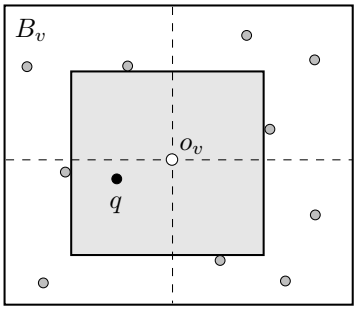


Figure 10: A bounded maximal P-empty rectangle of the subproblem at v.

In the following we therefore 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 (that is, P_v-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. [16] showed that there are $O(n_v)$ bounded maximal empty rectangles of this type associated with v. Suppose for specificity that this is the halfplane to the left of the y-axis; the other four cases are treated in a fully symmetric manner. See Figure 11. Consider the subset P_ℓ of points to the left of the y-axis. For each point p of P_ℓ there is (at most) a single rectangle in this family such that p is its left defining point. Similarly to the analysis in case (iv) of Section 4.1, we obtain this rectangle by connecting p to the y-axis by a horizontal segment, and shifting this segment up and down until it hits two other respective points of P_ℓ. Now we have a rectangle bounded by three points of P_ℓ whose right edge is anchored to the y-axis. Extend this rectangle to the right until its right edge hits a point to the right of the y-axis, to obtain the unique rectangle of this type with p on its left edge. (Here, and in the other cases discussed below, we assume that all the relevant points of P do exist; otherwise, the rectangle that we construct is not fully contained in the interior of B_v. This would be the case, for example, if

the shift of the above rectangle to the right of the y-axis does not encounter any point of P_v.) Clearly, there are $O(n_v)$ bounded maximal empty rectangles of this type.

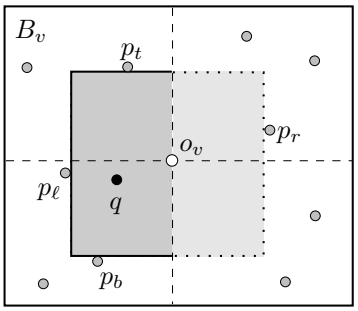


Figure 11: A bounded maximal P-empty rectangle with three defining points to the left of the y-axis.

We can find the part of each such rectangle which is to the left of the y-axis, by sweeping B_v with a vertical line from the y-axis to the left maintaining the points already seen in a balanced search tree, exactly as we computed the empty rectangles of type (iv) in Section 4.1. To find the part of each such rectangle on the right side of the y-axis we store the points to the right of the y-axis, sorted by their y-coordinates, in a search tree Σ_r. With each node of Σ_r we store the leftmost point stored in its subtree. We can identify the right edge of each rectangle R of the type under consideration by using Σ_r to find, in logarithmic time, the leftmost point to the right of the y-axis between the top and the bottom edges of R.

Overall we can find all rectangles of this type associated with v 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. This makes the size of S $O(n \log^3 n)$ if we store only the largest-area rectangle among all rectangles associated with each secondary node. The preprocessing time is $O(n \log^4 n)$ 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 12. (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

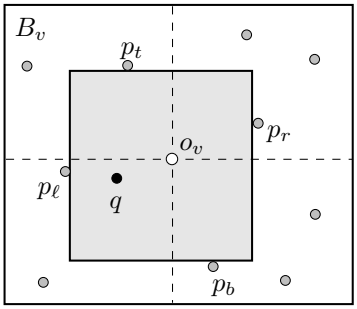


Figure 12: A bounded maximal P-empty rectangle with one defining point in each quadrant.

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 has to be 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 sweeps is not in the correct quadrant, one of the sweeps does not hit any point, or the last sweep does not hit p_r (e.g., because the point p_t is lower than p_r), 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. As we did for rectangles of type (i) (with three defining points in a halfplane), 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, as noted in the previous subcase.

4.3 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.)

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 assumed to be sorted from left to right (or, equivalently, from top to bottom).

Returning to our maximal P-empty rectangle R, we note that its defining points in the first (resp., third) quadrant are consecutive minimal points in F (resp., maximal points in E). There

are $O(n_v)$ such pairs in both lists.

Consider a consecutive pair (a, b) in E (with a to the left and above b). Let M_1 be the unique 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.

Similarly, let M_2 be the unique P-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 13. Our maximal P-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.

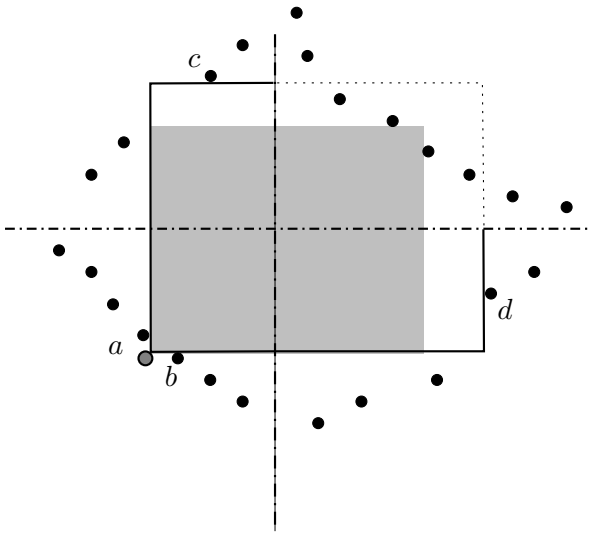


Figure 13: The structure of maximal empty rectangles with two defining points in each of the first and third quadrants. For each consecutive pair $\rho = (a, b)$ of maximal points in the third quadrant, there is an "interval" I_ρ of minimal points in the first quadrant such that any consecutive pair of points in I_ρ define with ρ a maximal empty rectangle. The point c in the second quadrant and the point d in the fourth quadrant define I_ρ.

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 P-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, ordered as above. This is done in $O(n_v \log n_v)$ time in the same way as we computed the chain of maximal points of P in Section 4.1. 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 14. It follows that in each column of A the defined entries also form a contiguous subsequence of rows. Therefore, A is a partial matrix. It is in fact more constrained than the general partial matrices defined in the introduction, because of the monotonicity of the defined portions of the rows and of the columns, as depicted in Figure 14. Aggarwal and Suri [4] call this specific form of a partial matrix a *double staircase* matrix.

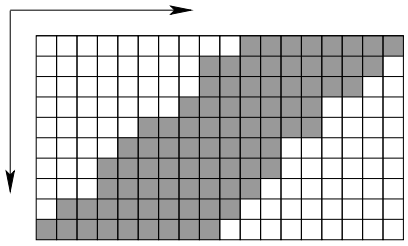


Figure 14: The structure of the defined portion of A (the arrows indicate the increasing order of rows and columns).

The following simple lemma, taken from [41], plays a crucial role in our analysis.

Lemma 4.1. [McKenna et al. [41]] *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 to the left and above x_2, and y_1 lies to the left and above 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*

$$A(y_1, x_1) + A(y_2, x_2) > A(y_1, x_2) + A(y_2, x_1). \tag{1}$$

Proof. The situation is depicted in Figure 15. 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. □

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

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

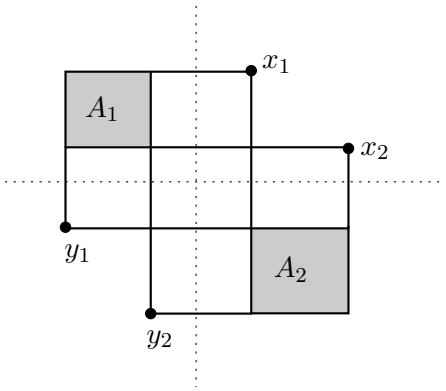


Figure 15: The inverse Monge property of maximal rectangles.

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

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

We separate the handling of a query into two cases, depending on which quadrant of B_v contains q.

(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. Note that in this case we do not need the full strength of the technique, as provided in Theorem 3.4, because the query submatrix is a "slab" consisting of complete consecutive columns, and the more efficient structure as provided by Theorem 3.6 (applied to the transpose of A) can be applied.

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 16.

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.

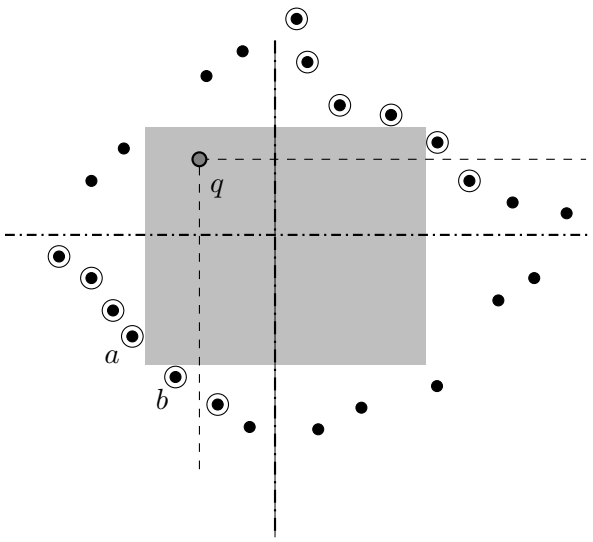


Figure 16: Querying with a point q in the second quadrant. The highlighted points form the prefixes E_q and F_q, plus one extra point in each chain.

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 the search set of q. (This can be reduced to $O(1)$ in case q is in the first or third quadrant of the subproblem defined by v, since in this case we can use, as already noted, the slab maximum data structure of Theorem 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 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 addition, we also have the two-dimensional segment tree S which stores $O(n\log^2 n)$ special maximal empty rectangles. As noted, the storage, preprocessing, and query costs for S are all dominated by the respective cost of the range-tree portion of the structure.

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

Theorem 4.2. *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.*

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 us to update the

weight of an edge, and to query for the distance between two arbitrary nodes u, v, namely, the smallest total weight of a path connecting u to v in G. The data structure requires linear storage and near-linear preprocessing, and answers a query or an update in $O(n^{2/3}\log^{5/3}(n))$ time.

Our data structure is based on the data structure of Klein [38], which only supports non-negative edge weights. We overcome this restriction by using *reduced costs* [33] which transform the problem into one with non-negative weights. Klein's data structure uses a data structure of Fakcharoenphol and Rao [24] that implements Dijkstra's algorithm efficiently. When using reduced costs, certain range minima data structures used in the data structure of Fakcharoenphol and Rao must be reconstructed after each weight update. We replace these components of the data structure with our row-interval minima data structure from Lemma 3.1 which has a faster construction time. We note that this idea, of using our new row-interval minima data structure in the data structure of Fakcharoenphol and Rao, has recently been used in the context of computing maximum flow in planar graphs [10]. The application to dynamic data structures for distance computations in planar graphs is new.

In Section 5.1 we describe Klein's data structure for distance queries on a dynamic planar graph with non-negative edge weights. In Section 5.2 we describe Fakcharoenphol and Rao's implementation of Dijkstra's algorithm [24] on the so called *Dense Distance Graph (DDG)* which is defined below. The basic data structure underlying their implementation, which we refer to as the *Monge heap*, is described in Section 5.3. With all this background handy, we describe in Section 5.4 how to modify Klein's data structure to allow negative edge weights. In Section 5.5 we show how to augment our data structure so that it can report the shortest path itself.

We make the following assumptions about G.

- We assume that G is simple, which can be ensured by removing self-loops and leaving only the shortest among each set of parallel edges.
- We assume that G is embedded in the plane (in the standard terminology, G is a *plane graph*). A plane embedding of G can be found in linear time (see, e.g., [31]).
- 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 will detect it (and abort the entire dynamic maintenance of G).

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 an edge-induced subgraph of G.[11] Fix some parameter $r < n$. An *r-division*, depicted in Figure 17, is a partition of E into $O(n/r)$ *pieces* such that the following additional properties hold.[12] (i) Each piece contains $O(r)$ nodes. We call a node which lies in more than one piece a *boundary node*. (ii) Each piece has $O(\sqrt{r})$ boundary nodes. Fredrickson [26] showed how to compute such an r-division in $O(n\log r + \frac{n}{\sqrt{r}}\log n)$ time, based on the Lipton-Tarjan planar separator theorem [40].[13]

Each piece may consist of multiple connected components. Consider a connected component C of some piece P and let C inherit its embedding from G. A face of a connected component C that is not a face in G is called a *hole* of C. (The hole of C is "filled" in G by other pieces. Typically, the outer face of C is a hole.) Note that, by definition, any boundary node in C lies on some hole of C. However, not every node on a hole of C is a boundary node (see Figure 17).

[11] That is the subgraph containing a particular set of edges and the vertices to which they are incident.

[12] Recall that our graph G is directed. In this discussion of r-division we refer to the underlying undirected graph $\overline{G}$ of G. We refer to connected components of $\overline{G}$ and apply the planar separator theorem to $\overline{G}$.

[13] Fredrickson did not require the pieces to be edge disjoint, but we can modify the r-division that his algorithm produces to be edge disjoint by associating each edge with a single arbitrary piece containing it.

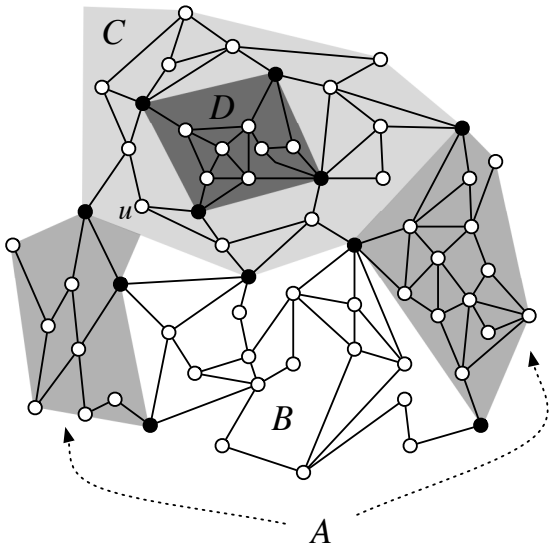


Figure 17: Illustration of an r-division. The edges of the graph are partitioned into four disjoint sets A, B, C, D indicated by different shades. Boundary nodes are indicated as solid. The piece A consists of two connected components, each with one hole (its unbounded face). Piece C consists of a single connected component that has two holes: the unbounded face, and the face inside which the piece D is embedded. Note that nodes that lie on a hole but do not have incident edges in more than one piece are not boundary nodes. For example, the node u lies on the unbounded face of the single connected component comprising C, but has no incident edges in any other piece. Hence u only belongs to C and is not a boundary node of C.

An *r-division with few holes* satisfies the following additional property: (iii) For every piece P and every connected component C of P, C has $O(1)$ holes. Italiano et al. [32] describe how to obtain an r-division with few holes in $O(n \log r + \frac{n}{\sqrt{r}} \log n)$ time using the cycle separator theorem of Miller [42].[14]

Let P be a piece of G. We denote by $d_P(u, v)$, for u, v nodes in P, the distance from u to v within P, that is, the smallest total weight of a path in P that connects u to v; $d_P(u, v) = \infty$ if there is no path from u to v in P.

5.1 Klein's dynamic algorithm

In this section we review the data structure of Klein [38] which supports distance queries between pairs of nodes and 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.

The structure uses the *dense distance graph* (DDG) of G, which is defined with respect to an r-division of G as follows. The nodes of the graph are the boundary nodes of the r-division, and its edges are pairs (u, v), where u and v are boundary nodes of the same connected component C of some piece P. The length of (u, v) is $d_P(u, v)$.[15] It follows that the DDG is the union of cliques, where each clique is composed of the boundary nodes of a connected component of a

[14] The pieces produced by this algorithm are edge disjoint.

[15] The DDG is in fact a multigraph since a pair of boundary nodes u and v may belong together to more than one piece (see Figure 17). It would be clear from the context which copy of the edge (u, v) we refer to.

single piece.

Fakcharoenphol and Rao [24] gave an implementation of Dijkstra's algorithm that runs on the DDG of G, for any fixed source vertex, in $O((n/\sqrt{r}) \log n \log r)$ time and uses $O(n)$ space. We describe the relevant details of this implementation in Section 5.2. This implementation requires, for each connected component of a piece, a data structure which we call, for short, a *Monge heap* (Fakcharoenphol and Rao call it an *on-line Monge searching* data structure). The construction of the Monge heaps of the connected components of a single piece takes $O(r \log r)$ time, and since there are $O(n/r)$ pieces, the construction time of the entire representation of the DDG is $O(n \log r)$ [38]. It follows that the total construction time of this static data structure (i.e., computing the r-division, and the Monge heaps, each representing the contribution of a connected component of some piece to the DDG. All these computations are done only once.) takes $O(n \log n)$ time for our choice of r (which would be $n^{2/3} \log^{2/3} n$).

The algorithm for answering a distance query from a node s in a piece P_s to a node t in a piece P_t consists of three steps:[16]

1. In the first step we find the distance inside P_s from s to every node of P_s, in $O(r \log r)$ time, using Dijkstra's algorithm. (This distance is ∞ for nodes of P_s outside the connected component containing s.) In particular we get the distance from s to every boundary node of P_s.[17] If $P_s = P_t$, that is, s and t are in the same piece, then this step computes in particular $d_{P_s}(s,t)$ (which of course does not have to be equal to the shortest distance between s and t). Note that we can skip this step if s is a boundary node of P_s.

2. In the second step we run an implementation of Dijkstra's algorithm, due to Fakcharoenphol and Rao [24], on the DDG, initializing the distance labels of the boundary nodes of P_s with their distances $d_{P_s}(s,\cdot)$ from s, as computed in the first step, and initializing the labels for all other nodes to ∞.[18] By doing this we simulate a single source shortest path computation in the DDG from an artificial source connected to each boundary node v of P_s by an edge whose weight is the distance from s to v in P_s. This implementation, described in Section 5.2, runs in $O((n/\sqrt{r}) \log n \log r)$ time and uses $O(n)$ space.

3. In the third step, we compute the distance from the boundary nodes of P_t to t inside P_t where the initial distance labels of the boundary nodes of P_t are their distances from s, as computed in the previous step. By doing this we simulate a single source shortest path computation inside P_t from an artificial source connected to each boundary node v of P_t by an edge whose weight is the distance from s to v as computed in Step (2). We implement this step by running Dijkstra's algorithm inside P_t with the appropriate initialization, which takes $O(r \log r)$ time. Note that we can skip this step if t is a boundary node of P_t.

Upon termination of the third step we can easily compute the weight ℓ of the shortest path from s to t among those containing a boundary node. If s and t are in different pieces the distance between them is ℓ, and if they are in the same piece P then the distance is $\min\{d_P(s,t), \ell\}$.

The correctness of the query algorithm is a consequence of the following property of shortest paths.

Lemma 5.1. *Assume that s and t belong to pieces, P_s and P_t, respectively. Let $s = v_0, v_1, \ldots, v_k = t$ be any shortest path from s to t. Then either $P_s = P_t$ and $s = v_0, v_1, \ldots, v_k = t$ is a path in*

[16] If s or t are boundary nodes, the choice of P_s or P_t, respectively, will not matter, as we indicate below.

[17] Klein [38] uses a multiple-source shortest-path data structure for this purpose, and finds the distances from s to all boundary nodes of P_s faster, in $O(\sqrt{r} \log r)$ time; however this does not affect the overall asymptotic cost of the algorithm.

[18] If s is a boundary node then we simply initialize its label to 0 and all other labels to ∞.

P_s, or there exists an increasing sequence of indices $0 \le i_0 < i_1 < \cdots < i_\ell \le k$ such that, for any $0 \le j \le \ell$, v_{i_j} is a boundary node, and the following holds.

1. *$v_0, v_1, \ldots v_{i_0}$ is a shortest path in P_s from s to v_{i_0}.*
2. *for any $0 \le j < \ell$, $v_{i_j}, v_{i_j+1}, \ldots, v_{i_{j+1}}$ is a shortest path from v_{i_j} to $v_{i_{j+1}}$ in some piece P. (In particular v_{i_j} and $v_{i_{j+1}}$ are boundary nodes of the same piece P.)*
3. *$v_{i_\ell}, v_{i_\ell+1}, \ldots v_k$ is a shortest path in P_t from v_{i_ℓ} to t.*

Proof. Follows immediately from the definition of boundary nodes and from the fact that, in the absence of negative-weight cycles, for any $0 \le i \le j \le k$, $v_i, v_{i+1}, \ldots, v_j$ is a shortest path from v_i to v_j. □

To update the weight of an edge e, we reconstruct from scratch the Monge heap associated with the (unique) connected component C of the piece P to which e belongs.

To conclude, a distance query to Klein's data structure takes $O(r \log r + (n/\sqrt{r}) \log n \log r)$ 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 $O(n^{2/3} \log^{5/3} n)$.

The query algorithm described here only works with non-negative edge weights because it uses an implementation of Dijkstra's algorithm to compute shortest paths. We show how to extend the algorithm for negative edge weights in Section 5.4.

5.2 Fast Dijkstra on the dense distance graph

We describe here the fast implementation of Dijkstra's algorithm on the dense distance graph (DDG) due to Fakcharoenphol and Rao [24, Section 3.2.2]. We run this procedure in the context of answering a distance query from some node s to another node t. We have initial distance labels for the nodes of the DDG as set in Step 2 of the query, described in the preceding subsection, and we want to run Dijkstra's algorithm on the DDG starting with these initial labels. Recall that Dijkstra's algorithm works as follows. It maintains a heap of the nodes, each with its distance label as the key. As long as there are nodes in the heap with a finite distance label, we extract the node u with the minimum label and *relax* all the edges emanating from u. That is, for each edge (u, v) we set $\delta(v) := \min\{\delta(v), \delta(u) + w(u, v)\}$, where $\delta(v)$ is the distance label of v and $w(u, v)$ is the weight of the edge (u, v). Upon termination, the distance label of each boundary node v is equal to the distance from s to v in G.[19]

By construction, the DDG has $O(\sqrt{r}(n/r)) = O(n/\sqrt{r})$ nodes and $O((\sqrt{r})^2(n/r)) = O(n)$ edges. A naive implementation of Dijkstra's algorithm therefore takes $\Omega(n)$ time, which is too slow for our purpose. The technique of Fakcharoenphol and Rao overcomes this difficulty by exploiting the Monge-like property of the edges of the DDG to avoid relaxing each edge explicitly (see below for details).

We use a separate Monge heap for each connected component of each piece. To simplify our presentation we will assume in the following description that each piece P is connected and thereby there is only one Monge heap associated with P. If P has multiple components we simply apply the exact same construction to each connected component separately.

The Monge heap data structure of a piece P, denoted as MH_P, maintains a distance label $\delta_P(v)$ for each boundary node v of P. At any time during the run of Dijkstra's algorithm, $\delta_P(v)$ is the length of the shortest path to v whose last edge is in P, among all such paths encountered so far. Note that, by definition, each boundary node v belongs to at least two distinct pieces,

[19] Dijkstra's algorithm also maintains a predecessor $\pi(v)$ for each vertex v to find the shortest path tree itself. It updates $\pi(v)$ to be u when a relaxation of an edge (u, v) changes $\delta(v)$.

and its labels within each piece may differ. MH_P supports the following operations. (To simplify the notation we usually do not write P as an explicit parameter of these functions. Each Monge heap has such a set of functions and it knows the piece P it is associated with.)

- FindMinInPiece(): Return a boundary node v in MH_P with minimum $\delta_P(v)$.
- ExtractMinInPiece(): Extract a boundary node v from MH_P with minimum $\delta_P(v)$.[20]
- ScanInPiece(v, d): Implicitly relax all the edges emanating from v in P, given that the true distance from s to v in G is d.[21] Recall that in the DDG all boundary nodes of P form a (bi-directional) clique. This operation implicitly sets $\delta_P(u) := \min\{\delta_P(u), d + d_P(v, u)\}$ for every boundary node u of P which is still in MH_P.

We also maintain a standard (global) heap H similar to the heap of the standard implementation of Dijkstra's algorithm. We denote by $\delta(v)$ the key of a node v in H. We store in H two kinds of nodes.

1. Nodes v with minimum distance labels in the Monge heaps. If v is the minimum of a Monge heap MH_P then we say that v is the *representative* of MH_P in H. Each node v is stored with a reference to MH_P, the Monge heap which it represents, and the key $\delta(v)$ of such a node v in H is its distance label in MH_P. (The same node v may appear in H multiple times with different keys as a representative of different Monge heaps.)
2. Boundary nodes of the piece containing s with initial finite distance label. The key $\delta(v)$ of such a node v in H is its initial distance label.

For every boundary node v we keep a bit indicating whether v was already visited or not. In contrast to standard implementations of Dijkstra's algorithm, we need to keep a record of whether a node v was already visited, since multiple copies of v may be stored in H. This happens since v has copies in more than one piece, and each such copy might be the representative of its piece in H. (Node v can also be in H if it is a boundary node of the piece of s with an initial finite distance label.) Moreover, due to the specific way in which the Monge heaps are implemented, as explained in the next subsection, there are $O(\log r)$ copies of each node in a single Monge heap; consequently a node v may become the minimum of this heap $O(\log r)$ times (and inserted as a representative into H). However, if we restrict the sequence of node extractions from H to the *first* extraction of each node then this subsequence of node extractions is identical to the sequence of node extractions performed by a traditional implementation of Dijkstra's algorithm. That is when a node v is first extracted from H, v is really the node with the minimum distance label picked by a traditional implementation of Dijkstra's algorithm. The other extractions are fictitious artifacts of the implementation and they when they occur.

Initially, for every boundary node v of the piece P containing s with initial distance label $d_P(s, v)$ we set $\delta(v) := d_P(s, v)$, and insert v with key $\delta(v)$ into H.

In each iteration we extract from H the minimum element. Let v be the node corresponding to the extracted element. If v is the minimum element of a Monge heap MH_P of a piece P then we perform ExtractMinInPiece() in P to delete this copy of v from MH_P. We find the new minimum in P by executing FindMinInPiece() of MH_P, and insert the new minimum into H.

If v is already marked as visited we do nothing else and continue to the next iteration. Otherwise, we mark v as visited and set $d(s, v) = \delta(v)$; that is, the key of v in H is the distance

[20] ExtractMinInPiece and FindMinInPiece return the same node in case of ties.

[21] A major property of Dijkstra's algorithm, also used in this implementation, is that it performs relax operations from a node v only after finding the true distance of v from s.

of v from s. For every piece P' containing v we perform ScanInPiece$(v, d(s, v))$ in $MH_{P'}$, followed by FindMinInPiece() in $MH_{P'}$. We update the element contributed to H by $MH_{P'}$ if it has changed or insert the minimum element of $MH_{P'}$ to H if this is our first operation on $MH_{P'}$.

As we show in the next subsection, FindMinInPiece() takes $O(1)$ worst-case time, ExtractMinInPiece() takes $O(\log r)$ worst-case time, and ScanInPiece() takes $O(\log^2 r)$ amortized time. Since the cost of a call to FindMinInPiece() is no larger than the cost of a call to ExtractMinInPiece() or to ScanInPiece(), and each time we call FindMinInPiece() we also call either ExtractMinInPiece() or ScanInPiece(), we can charge the total cost of the calls to FindMinInPiece() to the overall cost of the calls to ExtractMinInPiece() and to ScanInPiece().

A boundary node v of a piece P is returned $O(\log r)$ times by calls to ExtractMinInPiece() in P (see the next subsection for details), and we call ScanInPiece(v, d) once. Since there are $O(n/r)$ pieces, each with $O(\sqrt{r})$ boundary nodes, the total time spent on calls to these procedures is $O((n/\sqrt{r}) \log^2 r)$. However, the running time is dominated by the time spent on extracting the minimum elements from the global heap H.

Consider the size of the heap H at some specific time during the run of the algorithm. Each piece can contribute each of its boundary nodes to the global heap H at most once as the representative of the connected component containing it. Since there are $O(n/r)$ pieces, each with $O(\sqrt{r})$ boundary nodes the number of elements in H at any particular time is $O(n/\sqrt{r})$. It follows that the time to find the minimum element in H is $O(\log(n/\sqrt{r})) = O(\log n)$. Since the total number of boundary nodes is $O(n/\sqrt{r})$, and since each boundary node may be inserted into H $O(\log r)$ times (see the next subsection for details), the number of times we extract minimum elements from H is $O((n/\sqrt{r}) \log r)$. Hence the total running time of this variant of Dijkstra's algorithm on the dense distance graph is $O((n/\sqrt{r}) \log n \log r)$.

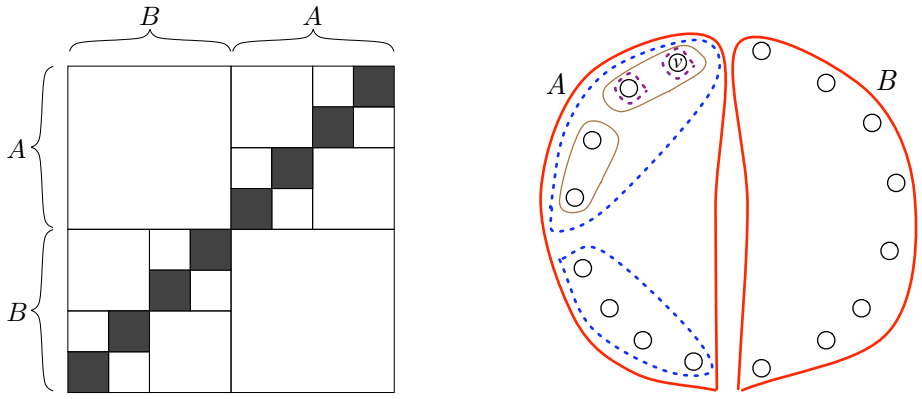
5.3 Monge heaps

In this subsection we describe the implementation of the Monge heap data structure MH_P of a piece P, given by Fakcharoenphol and Rao in [24, Section 4]. Recall that we simplify the presentation by assuming that each piece is connected. Otherwise, we construct a separate Monge heap, as described here, for every connected component of a piece. We start with a simple case in which (the connected component of) P has only one hole.

The first component of the Monge heap is a matrix M representing all edges contributed by P to the DDG. The rows (resp., columns), in the increasing order of their indices, correspond to the boundary nodes of P in clockwise (resp., counterclockwise) order around the boundary of the hole. The entry M_{uv} contains $d_P(u, v)$. In addition, for every boundary node u, Fakcharoenphol and Rao construct a range minima data structure over the row of u in M, to query for a node v minimizing $d_P(u, v)$ within a contiguous subsequence of boundary nodes along the boundary of the hole. Fakcharoenphol and Rao used a simple search tree-based range minima structure for this purpose.

Because of the cyclic nature of the order of the nodes along the hole boundary, the matrix M does not necessarily have the Monge property. This however can be fixed, as done by Fakcharoenphol and Rao, by splitting M into disjoint submatrices, such that each submatrix does have the Monge property. Specifically, the boundary nodes of P are split into two contiguous sequences A and B of (roughly) equal length. Then the submatrix of M whose rows correspond to the nodes of A and whose columns correspond to the nodes of B is a Monge matrix. (This is argued in much the same way as in Monge's original observation, mentioned in the introduction.) Similarly, the submatrix of M whose rows correspond to the nodes of

B and whose columns correspond to the nodes of A is also a Monge matrix. We recursively split the remaining submatrices $A \times A$ and $B \times B$ in the same way, until we get to sequences of length 1. See Figure 18 for an illustration. Together, all the submatrices defined by this recursive partitioning of M are pairwise disjoint and cover M. Note that each boundary node is contained in $O(\log r)$ submatrices.



(a) Submatrices resulting from the first three levels of the recursive decomposition are shown. The shaded blocks require further decomposition.

(b) Illustration of the corresponding partitions of boundary nodes. All bipartitions in which a particular node v appears are indicated.

Figure 18: Illustration of the decomposition of M into Monge Submatrices.

The initialization of the Monge heap consists of computing (explicitly, all entries of) M, splitting it into submatrices, and constructing the range minima data structures. Since all boundary nodes lie on a single hole, computing the entries of M (i.e., distances between all pairs of boundary nodes) takes $O(r \log r)$ time and $O(r)$ space using Klein's multiple-source shortest-path algorithm [38] (see also [14]). It can be easily verified that all the other tasks can be performed within the same time and space bounds. This entire process is done once, during preprocessing, before handling any query or update.

During a distance computation we maintain, for each submatrix M', a *bipartite Monge subheap* (Fakcharoenphol and Rao call it a *bipartite on-line Monge searching* data structure), which supports the Monge heap operations restricted to the bipartite subgraph that M' represents, using the fact that the submatrix has the Monge property.

The bipartite Monge subheap. The bipartite Monge subheap H' of a Monge submatrix M' with a set of rows A and a set of columns B maintains a distance label $\delta'(u)$ for every node $u \in A$. Initially $\delta'(u) = \infty$ for every $u \in A$. The distance label of a node $v \in B$ is defined to be $\delta'(v) = \min_{u \in A}\{\delta'(u) + M'_{uv}\}$ (recall that $M'_{uv} = d_P(u, v)$) and is not stored explicitly. For every node $v \in B$ we maintain a bit indicating whether v has already been extracted from H' or not. Initially all nodes of B are in H'. A node v is extracted from H' when it has the smallest distance label among all nodes in the global heap H and therefore also in the Monge heap MH_P containing H'. The monotonicity of the smallest label in Dijkstra's algorithm implies that after v is extracted from H' a subsequent decrease in $\delta'(u)$ for a node $u \in A$ cannot cause $\delta'(v)$ to decrease: the value of $\delta'(u)$ (which will always be lower bounded by $d(s, v)$) cannot be smaller than the value of $\delta'(v)$ when it was extracted from H'.

The bipartite Monge subheap structure supports the following operations (H' is implicit in the following notation):

- FindMin(): Return a node $v \in B$ with minimum distance label $\delta'(v)$ among those that have not yet been extracted.
- ExtractMin(): Extract a node $v \in B$ with minimum distance label $\delta'(v)$ among those that have not yet been extracted.[22]
- Scan(u, d): Set $\delta'(u) = d$ for $u \in A$, and (implicitly) relax all the edges from u to the nodes of B.

We say that $u \in A$ is the *parent* of $v \in B$ (and v is the *child* of u) if $\delta'(u)$ is finite and $\delta'(v) = \delta'(u) + M'_{uv}$; see below for handling ties. As the execution of Dijkstra's algorithm progresses, the distance labels $\delta'(u)$ of nodes $u \in A$ may change and thereby the parents of nodes $v \in B$ may change. Assuming that the parent of every member of $v \in B$ is uniquely defined (at some fixed time), the Monge property of M' implies the following two properties:

1. The set of nodes of B for which a specific node $u \in A$ is the parent are consecutive in B.
2. If node u' follows u in A, then the set of nodes of which u' is the parent follows in B the set of nodes of which u is the parent.

Furthermore, in case of ties, i.e. when there are two different nodes $u_1, u_2 \in A$ such that $\delta'(v) = \delta'(u_1) + M'_{u_1 v} = \delta'(u_2) + M'_{u_2 v}$, the Monge property of M' implies that we can always break the tie and pick a parent for v such that Properties (1) and (2) above continue to be satisfied. Our algorithm will indeed (implicitly) maintain parents in this manner such that Properties (1) and (2) are satisfied. Note that nodes in B with infinite distance labels will also have parents in A. For such a node v any node $u \in A$ satisfies that $\delta'(v) = \delta'(u) + M'_{uv}$ and we can pick any node to be a parent of v as long as we maintain Properties (1) and (2).

We maintain a binary search tree T of triplets of the form (a, b_1, b_2) where $a \in A$, $b_1, b_2 \in B$, and the set of nodes of B between b_1 and b_2 (inclusive) is a maximal interval of nodes of B that are currently in H' and whose parent is a. See Figure 19. Note that the same node $a \in A$ may appear in more than one triplet of T because, after extracting nodes of B from the bipartite Monge subheap, the set of non-extracted nodes of which a is a parent may not be a contiguous subsequence of the nodes of B.[23] That is, the extracted elements form gaps in the sequence, so we need to record the remaining elements as the union of smaller contiguous subsequences. The order of the triplets in T is lexicographic. Namely: (1) If node a' follows a in A then all the triplets (a', b'_1, b'_2) follow all the triplets (a, b_1, b_2). (2) The set of triplets (a, b_1, b_2) of the same node a are ordered in T by the order of their (pairwise disjoint) intervals in B.

The bipartite Monge subheap structure also consists of a standard heap H_B containing, for every triplet $(a, b_1, b_2) \in T$, a node b between b_1 and b_2 (inclusive) that minimizes $\delta'(b) = \delta'(a) + M'_{ab}$. The key of b in H_B is $\delta'(b)$.[24]

The operations on a bipartite Monge subheap are implemented as follows:

- FindMin(): Return a node b with minimum distance label in H_B.
- ExtractMin(): Extract the node b with minimum distance label from H_B. We mark b as removed from H'. We find the (unique) triplet (a, b_1, b_2) containing b in T. Let b' and b'' be the members of B that precede and follow b, respectively, within this triplet,

[22] As before, ExtractMin() and FindMin() return the same node in case of ties.

[23] For example, as depicted in Figure 19, z may be the parent of nodes 9–16 of B just before we extract node 13 of B from the subheap. After this extraction z is the parent of nodes 9–12 of B which form one triplet $(z, 9, 12)$, and of nodes 14–16 which form another triplet $(z, 14, 16)$.

[24] This heap can be implemented within the tree T by maintaining subtree minima at the nodes of T.

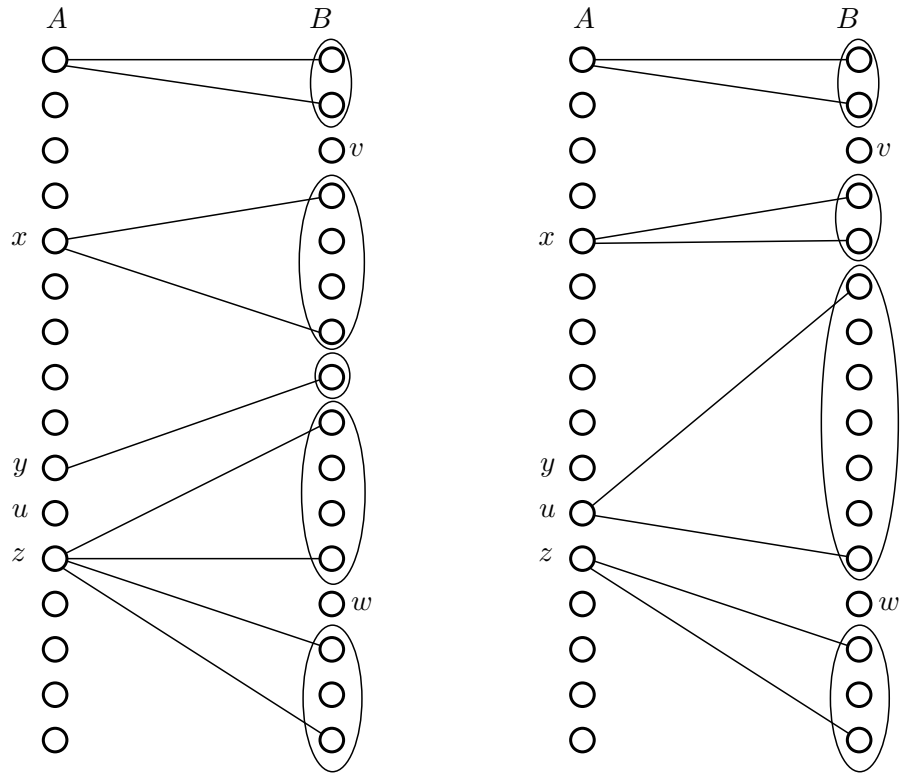


Figure 19: The triplets in the tree T of a bipartite Monge subheap. Each triplet (a, b_1, b_2) is depicted by a line from a to b_1, a line from a to b_2 and an ellipse around the nodes of B between b_1 and b_2. On the left we see the triplets before the scan of u and on the right we see the triplets after that scan. Nodes v and w have already been deleted from this bipartite Monge subheap and therefore do not belong to any triplet.

if they exist. We split the triplet (a, b_1, b_2) and replace it with two triplets (a, b_1, b') and (a, b'', b_2) (if these intervals are defined). In each of these new triplets we find the node b^* that minimizes $\delta'(b^*) = \delta'(a) + M'_{ab^*}$, using the range minima data structure for the row of a in M', and then we insert b^* into H_B.

- SCAN(u, d): First we set $\delta'(u) = d$ and then we find the children of u in B.

  If u is the first node in the bipartite Monge subheap for which this operation is applied then all the nodes of B are children of u (none of them could have already been removed, and u is the only vertex in A with a finite distance label). Otherwise, we next describe how to find the children of u which are in triplets associated with nodes preceding u in A. We find children of u which are in triplets associated with nodes following u in A in a fully symmetric manner.

  If there is no triplet $t = (w, f_1, f_2)$ such that w precedes u in A then u does not have children in triplets associated with nodes preceding it in A. Otherwise, we search T and find the last triplet $t = (w, f_1, f_2)$ such that w precedes u in A. We then traverse the triplets in T backward, starting from t, until we reach a triplet (a, b_1, b_2) such that $\delta'(a) + M'_{ab_1} < \delta'(u) + M'_{ub_1}$ or until we scanned all triplets preceding t without finding such a triplet (a, b_1, b_2). Then we perform one of the following steps.
  (1) If we did not find a triplet (a, b_1, b_2) as above then all nodes in triplets preceding t become children of u.

(2) If $\delta'(a) + M'_{ab_2} \geq \delta'(u) + M'_{ub_2}$ then among nodes in triplets preceding t, the first node in B whose parent changes to u belongs to the triplet (a, b_1, b_2). We find it by a binary search on the subsequence of B between b_1 and b_2.
(3) If $\delta'(a) + M'_{ab_2} < \delta'(u) + M'_{ub_2}$ and (a, b_1, b_2) is not t then among nodes in triplets preceding t, the first node in B whose parent changes to u is the first node in the triplet following (a, b_1, b_2).
(4) If $\delta'(a) + M'_{ab_2} < \delta'(u) + M'_{ub_2}$ and (a, b_1, b_2) is t then none of the nodes in triplets associated with nodes preceding u in A acquires u as its parent.

Notice that, as mentioned, the Monge property implies that the sequence of children that u acquires in B is contiguous. Moreover, since Dijkstra's algorithm finds the distances in monotonically increasing order, and since we extract a node b from H' only when b is the global minimum in the global heap H of Dijkstra's algorithm, then once b is removed from H' it cannot acquire a new parent (as that would mean that we discovered a shorter path to b). These two observations imply that if there are two consecutive triplets (w, x, y) and (w', x', y') such that w' precedes w in A and there is a vertex $z \neq y', x$, already extracted from H', in between y' and x in B, then the search for children of u, while scanning u as described above, will not proceed beyond (w', x', y'). For an example look at the scan of u in Figure 19. Following this scan u acquires the children of z from one of the two triplets of z, the child of y, and two out of the four children of x. Since v and w have already been deleted, then u cannot acquire children preceding v or following w.

Let x (resp., y) be the first (resp., last) child of u in B, as obtained in the preceding step. We remove from T all triplets containing nodes between x and y, and remove from H_B the elements that these triplets contributed to H_B. Let (a, b_1, b_2) be the triplet containing x. If $x \neq b_1$ then we shorten this triplet to (a, b_1, z_1) where z_1 is the node preceding x in B. Similarly, we shorten the triplet (a', b'_1, b'_2) containing y to (a', z_2, b'_2) where z_2 is the node following y in B, if $y \neq b'_2$. We form a new triplet (u, x, y) and insert into T the triplets (a, b_1, z_1), (u, x, y), and (a', z_2, b'_2) in this order. Finally, we update the nodes of B that these new triplets contribute to H_B. We find each of these nodes by a range minimum query in the range minima data structure of the appropriate row of M.

Clearly, FindMin() takes $O(1)$ time. It is also easy to verify that ExtractMin() takes $O(\log r)$ (worst-case) time and Scan() takes $O(\log r)$ amortized time: Both ExtractMin() and Scan() insert a constant number of new triplets to T in $O(\log r)$ time. Scan(u, d), however, may traverse many triplets to identify the children of u, but it deletes all except at most two of the triplets that it traverses. Therefore we can charge the traversal and deletion of these triplets to their previous insertion in a previous ExtractMin() or Scan().

Implementation of the Monge heap. Now we get back to the implementation of the Monge heap itself. Recall that the Monge heap data structure of a piece P maintains a distance label $\delta_P(v)$ for each boundary node v of P. At any time during the run of Dijkstra's algorithm, $\delta_P(v)$ is the length of the shortest path to v encountered so far, that reaches v through an edge of the piece P.

The Monge heap MH_P of P contains a bipartite Monge subheap for each submatrix of M, and a heap H_M containing the minima of the heaps H_B of the bipartite Monge subheaps of P. (Note that the same node may appear in H_M multiple times, possibly with different keys, if it is the minimum in several bipartite Monge subheaps.) Each time the minimum of a heap H_B changes, we also update the element that it contributes to H_M. Note that we do not store $\delta_P(v)$ explicitly for any node v; instead $\delta_P(v)$ is implicitly represented as the minimum $\delta'(v)$ over all bipartite Monge subheaps containing v.

The implementation of the operations is as follows:

- FindMinInPiece(): Return the minimum in H_M.
- ExtractMinInPiece(): Extract the minimum node from H_M, and perform ExtractMin() in the bipartite Monge subheap that contributed this element to H_M. Then we update H_M with the new minimum of that bipartite Monge subheap.
- ScanInPiece(v, d): We perform Scan(v, d) in each of the $O(\log r)$ bipartite Monge subheaps containing elements of the row of v. After the scans we update H_M as necessary.

Note that ExtractMinInPiece() extracts the node with the minimum distance label only from the bipartite Monge subheap where this minimum is obtained. Since there are $O(\log r)$ copies of each node in the bipartite Monge subheaps, each node is extracted $O(\log r)$ times. Each extraction from a bipartite Monge subheap takes $O(\log r)$ time. In contrast, ScanInPiece(v, d) scans all the bipartite Monge subheaps containing v,[25] and it thus takes $O(\log^2 r)$ amortized time.

Dealing with multiple holes. We now treat the case where the boundary nodes of a piece are incident to more than one hole. We keep a distance matrix, range minima structures, and a corresponding set of bipartite Monge subheaps, as above, for each hole separately. The heap H_M of MH_P contains the minima of all bipartite Monge subheaps of all the holes. This handles all edges of the DDG whose endpoints lie on a common hole.

However, the DDG also has edges whose endpoints lie on two distinct holes. For each ordered pair of holes we form a matrix associated with the edges between the nodes on the first hole and the nodes on the second hole. Consider a particular pair of holes and the associated matrix M. The rows of M correspond to nodes on one hole, say h_1 and the columns correspond to nodes of the other hole, say h_2. Although M is derived from a bipartite graph, the bipartition corresponds to nodes on the boundary of two distinct faces (holes) in the underlying planar graph. Therefore M is not Monge in general. Fakcharoenphol and Rao observed that we can still use a (single) bipartite Monge subheap to represent M by modifying it so that B is cyclically ordered and a triplet in T can wrap around and contain a suffix of the nodes of B followed by a prefix of the nodes of B.

We describe here a different solution to this issue. The advantage of this solution is that it only involves Monge matrices and does not require changing the Monge heap itself. Hence, the use of our new row-interval minima data structure that we discuss in the next section easily applies to the case of multiple holes as well. As shown in [45, Section 4.4], one can compute from h_1, h_2, and the underlying planar graph, in $O(r \log r)$ time, two Monge matrices M^ℓ and M^r whose rows correspond to vertices of h_1 and whose columns correspond to vertices of h_2. The matrices M^ℓ and M^r satisfy that $\min\{M^\ell_{ij}, M^r_{ij}\} = M_{ij}$, that is, the minimum of M^r_{ij} and M^ℓ_{ij} is the length of the arc from i to j in the DDG for every pair i, j of indices. For each pair of holes of P, we keep the above pair of surrogate matrices, along with the range minima structures and bipartite Monge subheaps for each of them as part of MH_P. We also add the minima of these two bipartite Monge subheaps to H_M and operate on them as on the other bipartite Monge subheaps. Since the number of holes in P is $O(1)$, the time complexity and the space requirements remain asymptotically the same as in the case of a single hole.

[25] We do not extract the scanned node v from all Monge heaps and bipartite Monge heaps containing it but only from the one in which v has the smallest label. In the other heaps v may have a larger label, and therefore it can still get a new parent. While this will never result in a smaller overall label for v, maintaining the correct parent at each bipartite Monge subheap is crucial for the integrity of the data structure.

5.4 Negative edge weights

With all the machinery presented so far in this section, we now proceed to extend Klein's data structure, which was described in Section 5.1, so as to allow negative edge weights. We deal with negative edge weights using *reduced costs* [33]. 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 $d(u, v)$, then the distance between the nodes with respect to the reduced costs is $d(u, v) + \phi(u) - \phi(v)$.

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. This is an immediate consequence of the triangle inequality for distances.

The advantage of using reduced costs is that it allows us to run Dijkstra's algorithm (which requires non-negative edge weights) rather than less efficient algorithms (such as the Bellman-Ford algorithm).

Consider a run of Dijkstra's algorithm for finding distances from some given source node s with some initial distance labels $\delta(v)$ for each node v (as in steps (2) and (3) of our query algorithm). With reduced costs, corresponding to a feasible price function ϕ, we initialize, for each node v, the distance label of v to $\delta(v)+\phi(s)-\phi(v)$ rather than to $\delta(v)$. This is the correct initialization since this is the value that the initial distance label of v would have been if we had run the previous stages with reduced costs corresponding to ϕ. After Dijkstra's algorithm terminates, the distance from s to a node u is $\delta(u)+\phi(s)-\phi(u)$. Equivalently, we may initialize the distance label of v to be $\delta(v) - \phi(v)$. In this case when Dijkstra's algorithm terminates the distance from s to a node u is $\delta(u)-\phi(u)$. This does not change the run of Dijkstra's algorithm as we just subtracted $\phi(s)$ from the distance labels of all nodes. Note that distance labels in such a run of Dijkstra's algorithm may be negative, but as long as the reduced costs themselves are (implicitly) non-negative the output of Dijkstra's algorithm is correct.

In our setting we use two types of price functions. First, for every connected component C of a piece P we have a price function ϕ_C that is used to compute distances inside C (in the first and the third steps of a query). Second, we have a price function ϕ_{dd} for the dense distance graph DDG. Note that the prices of the same node v according to each of its associated price functions (one for each piece containing v and one for the DDG) may be different.

Initially, at the preprocessing stage, we choose an arbitrary node v, and compute the distances from v to all other nodes, using a single source shortest path algorithm for planar graphs that can handle negative weights, in $O(n \log^2 n/ \log\log n)$ time [45]. This cost dominates the construction time of the data structure. We use these distances as the values of the price functions ϕ_C for every connected component C of a piece P, and of ϕ_{dd}. It is easy to see that, initially, both ϕ_C and ϕ_{dd} are feasible price functions (i.e., they induce non-negative reduced costs). Later on, we will have to update these price functions when edge weights are updated; see below for details.

We modify the query algorithm so that it uses the reduced costs rather than the original weights. In the first step of the query, we run Dijkstra's algorithm inside the connected compnent C_s containing s of the piece P_s and find the distances from s to all nodes of C_s. As described in general before, we run Dijkstra's algorithm with the label of s initialized to $-\phi_{C_s}(s)$ (we subtract $\phi_{C_s}(s)$ from all the distance labels) and with the reduced costs with respect to ϕ_{C_s}. This yields distance labels $\delta(v)$ for all $v \in C_s$. We restore the correct distances by putting $d_{C_s}(s, v) := \delta(v) + \phi_{C_s}(v)$, for every node $v \in C_s$.

In the second step we compute the distance from the boundary nodes of C_s to all boundary

nodes in the DDG, using reduced costs (with respect to ϕ_{dd}), by the implementation of Dijkstra's algorithm given in Section 5.2. As described before, we initialize the distance label of each boundary node v of C_s to $d_{C_s}(s,v) - \phi_{dd}(v)$. When this run of Dijkstra terminates we recover, for each boundary node u of the connected component C_t containing t, of the corresponding piece P_t, the distance from s to u by adding $\phi_{dd}(u)$ to its computed distance label. We run the third step of the query in a similar manner, using reduced costs with respect to ϕ_{C_t}.

Note that the price function ϕ_{dd} may change during updates of edge weights. However, we cannot afford to explicitly compute the reduced costs of the edges of the DDG when processing a query or update, since that would take $\Omega(n)$ time. Instead, whenever the implementation requires the reduced cost of an edge (u,v) in the DDG, we compute it on the fly by adding $\phi_{dd}(u) - \phi_{dd}(v)$ to $d_C(u,v)$, where C is the relevant connected component of some piece P.

A critical technical issue that requires attention is that we need to recompute the range minima data structures required by Fakcharoenphol and Rao's implementation of Dijkstra's algorithm, so that they can answer range minimum queries with respect to reduced costs rather than original weights. This recomputation is required whenever the price function ϕ_{dd} changes (due to updates of edge weights). Reconstructing the row range minima structures as done by Fakcharoenphol and Rao would take $\Omega(n)$ time. Instead, our main contribution, as mentioned in the beginning of this section, is to replace these structures with the row-interval minima data structure of Lemma 3.1 for each Monge submatrix M'. As we show below, the overall construction (and reconstruction) time of these data structures is faster, so we can efficiently maintain them after each change in the price function.[26]

We bound the time it takes to compute all these data structures by considering each of the Monge submatrices separately. Computing the row-interval minima data structure of Lemma 3.1 for a matrix with x rows and columns takes $O(x \log x)$ time. This sums to $O(\sqrt{r} \log^2 r)$. over all the submatrices of the Monge heaps of the connected components of a single piece (and their holes). Summing over all n/r pieces we get that the total construction time for all of the row-interval data structures is $O((n/\sqrt{r}) \log^2 r)$, which is dominated by 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 DDG, we compute these row-interval minima data structures with respect to the initial price function defined above.

Now consider an update of the weight of an edge $e = (u,v)$ in a connected component C of 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, which we report and halt. Otherwise, we observe that the current distances from v can be used as a feasible price function both before and after the weight of e is updated, since a shortest path from v to any node does not have to use e (because e would create a cycle in the path with non-negative weight). We compute the distances from v in C as in the first step of the query algorithm and use them as the values of the new price function ϕ_C for the connected component C. Next, we compute distances from v to every boundary 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 C and to all boundary nodes is $O(r \log r + (n/\sqrt{r}) \log n \log r)$, which is the time bound for a distance query. The change in the weight of e may change the distances inside C. We therefore recompute the distances between

[26]It is easy to verify that our Monge submatrices maintain their Monge property even when we measure distances with respect to reduced costs rather than costs — adding a fixed value to each row and a fixed value to each column has no effect on the Monge property. Therefore the correctness of the algorithm is maintained and we can indeed use the data structure of Lemma 3.1.

all boundary nodes of C inside C in $O(r \log r)$ time and update the distance matrices of the connected component C, as in the data structure for non-negative edge-weights described in Section 5.1. We recompute these distances using Klein's multiple-source shortest-path algorithm [38] and we use the distances from v in C as a feasible price function for Klein's algorithm which requires non-negative edge weights. Since we have updated the price function of the dense distance graph we must recompute all row-interval minima data structures. This takes $O((n/\sqrt{r}) \log^2 r)$ time, as argued above.

We obtain that the running time for a query or an update remains $O(r \log r + (n/\sqrt{r}) \log n \log r)$. Choosing $r = n^{2/3} \log^{2/3} n$ yields a bound of $O(n^{2/3} \log^{5/3} n)$ time per query and update. We conclude with the following theorem which summarizes the result presented in this section.

Theorem 5.2. *Given a possibly negatively weighted directed planar graph G with n vertices, we can construct a data structure that supports distance queries in G between arbitrary pairs 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.*

5.5 Reporting the shortest path

We finally show how to augment our data structure so that it can also report the shortest path Q itself with an additional $O(|Q| \log \log \Delta)$ additive overhead to the query time. Here Δ is the maximum degree of a node in the path Q.

A standard implementation of Dijkstra's algorithm maintains for each node v a *predecessor* node u, that "gave" the current distance label to v. That is, the node u such that $\delta(v) = \delta(u) + w(u, v)$. Using this information, it is easy to start from t and backtrack from a node to its predecessor along the shortest path from s to t. We describe a similar mechanism for the Monge heaps in the fast Dijkstra implementation on the dense distance graph.

In the implementation of Dijkstra's algorithm on the DDG, a node v is contained in several Monge subheaps. Consider one such Monge subheap H' with set of rows A and set of columns B. The predecessor in H' of a node $v \in B$ is its parent $u \in A$. The predecessor of a node v on the shortest path from s to v is its parent in the Monge subheap H' from which it is first extracted when it is the minimum of the global heap H for the first time (as a representative of some Monge heap MH_P containing H'). We record the parent of v in the triplet containing it in H' when it is extracted from H'.

Now, we can backtrack along the shortest path Q from s to t. We begin at t and backtrack along the suffix of Q in P_t. We do this using regular predecessor pointers saved by the run of Dijkstra's algorithm inside P_t. Then, we backtrack along the part of Q within the DDG that was found by the second step of the query algorithm. We start from the last vertex of Q on the boundary of P_t and proceed to the first vertex of Q on the boundary of P_s in the DDG, using the predecessor pointers recorded by the run of Dijkstra's algorithm on the DDG as described above. Last, we backtrack along the prefix of Q inside P_s using predecessor pointers recorded by the run of Dijkstra's algorithm inside P_s.

The backtracking along Q inside P_t and P_s gives us the complete suffix of Q inside P_t and the complete prefix of Q inside P_s, respectively. The backtracking in DDG, on the other hand, gives us only the boundary nodes along the middle part of Q. The recovery of this partial information about Q does not add to the asymptotic running time of the query.

To recover all the nodes along the middle part of Q, we proceed as follows. Recall that we obtained the distance matrix of each hole and the distance matrices of each pair of holes of each connected component C of a piece P by applying Klein's multiple-source shortest-path algorithm to C [38, 45]. This multiple-source shortest-path algorithm can produce a data structure of size linear in C that, for any node v, and any boundary node s, can return the first edge on the

shortest path from v to s in $O(\log\log\Delta)$ time. In particular we can use this data structure to report the path R which defines the weight of an edge (u, v) in the DDG in $O(|R|\log\log\Delta)$ time, and therefore we can use this structure to report the path Q in $O(|Q|\log\log\Delta)$ time, as required.

We have thus established the following extension of Theorem 5.2.

Theorem 5.3. *Given a possibly negatively weighted directed planar graph G with n vertices, we can construct a data structure that supports distance queries in G between arbitrary pairs of nodes and updates to edge weights, in $O(n^{2/3}\log^{5/3} n)$ time per operation, as well as shortest path queries between pairs of nodes in $O(n^{2/3}\log^{5/3} n + |Q|\log\log\Delta)$ time per query, where Q is the reported shortest path, and Δ is the maximum degree in G of a node in Q. The data structure can be constructed in $O(n\log^2 n/\log\log n)$ time and requires $O(n)$ space.*

References

[1] Aggarwal, A., and Klawe, M. Applications of generalized matrix searching to geometric algorithms. *Discrete 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. (FOCS)* (1988), pp. 497–512.

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

[5] Atallah, M. J., and Frederickson, G. N. A note on finding a maximum empty rectangle. *Discrete 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, pp. 820–825.

[8] Bender, M. A., and Farach-Colton, M. The LCA problem revisited. In *Proc. 4th Latin Amer. 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 Canad. 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. Found. Comput. Sci. (FOCS)* (2011), pp. 170–179.

[11] Brodal, G. S., Davoodi, P., and Rao, S. S. On space efficient two dimensional range minimum data structures. In *Proc. 18th Eur. Symp. Algorithms (ESA)* (2010), vol. 6347 of *LNCS*, Springer, pp. 171–182.

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

[13] Cabello, S. Many distances in planar graphs. *Algorithmica 62*, 1-2 (2012), 361–381.

[14] Cabello, S., and Chambers, E. W. Multiple source shortest paths in a genus g graph. In *Proc. 18th Annu. ACM-SIAM Symp. on Discrete Algorithms* (2007), pp. 89–97.

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

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

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

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

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

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

[21] Dehne, F. Computing the largest empty rectangle on one-and two-dimensional processor arrays. *J. Parallel. Distr. Comput. 9*, 1 (1990), 63–68.

[22] 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.

[23] Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. Making data structures persistent. *J. Comput. Syst. Sci. 38*, 1 (1989), 86–124.

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

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

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

[27] Gutiérrez, G., and Paramá, J. R. Finding the largest empty rectangle containing only a query point in large multidimensional databases. In *Proc. of the 24th Int. Conf. Sci. Stat. Database Manag. (SSDBM)* (2012), pp. 316–333.

[28] Harel, D., and Tarjan, R. E. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput. 13*, 2 (1984), 338–355.

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

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

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

[32] 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. (STOC)* (2011), pp. 313–322.

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

[34] Kaplan, H., Mozes, S., Nussbaum, Y., and Sharir, M. Submatrix maximum queries in Monge matrices and Monge partial matrices, and their applications. In *Proc. 23rd Annu. ACM-SIAM Symp. on Discrete Algorithms* (2012), pp. 338–355.

[35] Kaplan, H., and Sharir, M. Finding the maximal empty disk containing a query point. In *Proc. 28th Annual ACM Symp. Comput. Geom. (SOCG)* (2012), pp. 287–292.

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

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

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

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

[40] Lipton, R. J., and Tarjan, R. E. Applications of a planar separator theorem. *SIAM J. Comput. 9*, 3 (1980), 615–627.

[41] 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.

[42] Miller, G. L. Finding small simple cycle separators for 2-connected planar graphs. *J. Comput. Syst. Sci. 32*, 3 (1986), 265–279.

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

[44] Mozes, S., and Sommer, C. Exact distance oracles for planar graphs. In *Proc. 23rd Annu. ACM-SIAM Symp. on Discrete Algorithms (SODA)* (2012), pp. 209–222.

[45] 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. Algorithms (ESA)* (2010), M. de Berg and U. Meyer, Eds., vol. 6347 of *LNCS*, Springer-Verlag, pp. 206–217.

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

[47] 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.

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

[49] Orlowski, M. A new algorithm for the largest empty rectangle problem. *Algorithmica 5*, 1 (1990), 65–73.

[50] Overmars, M. H., and van Leeuwen, J. Maintenance of configurations in the plane. *J. Comput. Syst. Sci. 23*, 2 (1981), 166–204.

[51] 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.

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

[53] Yuan, H., and Atallah, M. J. Data structures for range minimum queries in multidimensional arrays. In *Proc. 21st Annu. ACM-SIAM Symp. on Discrete Algorithms (SODA)* (2010), pp. 150–160.