Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1989

Parallel Rectilinear Shortest Paths with Rectangular Obstacles

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Danny Z. Chen

Report Number:

89-889

Atallah, Mikhail J. and Chen, Danny Z., "Parallel Rectilinear Shortest Paths with Rectangular Obstacles" (1989). *Department of Computer Science Technical Reports.* Paper 757.
https://docs.lib.purdue.edu/cstech/757

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

PARALLEL RECTILINEAR SHORTEST PATHS WITH RECTANGULAR OBSTACLES

Mikhail J. Atallah
Danny Z. Chen

CSD-TR-889
July 1989
(Revised March 1991)

Parallel Rectilinear Shortest Paths with Rectangular Obstacles

Mikhail J. Atallah* Danny Z. Chen†

Abstract

Given a rectilinear convex polygon P having $O(n)$ vertices and which contains n pairwise disjoint rectangular rectilinear obstacles, we compute, in parallel, a data structure that supports queries about shortest rectilinear obstacle-avoiding paths in P. That is, a query specifies a source and a destination, and the data structure enables efficient processing of the query. We construct the data structure in $O(\log^2 n)$ time, with $O(n^2/\log^2 n)$ processors in the CREW-PRAM model if all queries are such that the source and the destination are on the boundary of P, with $O(n^2/\log n)$ processors if the source is an obstacle vertex and the destination is on the boundary of P, and with $O(n^2)$ processors if both the source and destination are arbitrary points in the plane. The data structure we compute enables one processor to obtain the path length for any pair of query vertices (of obstacles or of P) in constant time, or $O(\lceil k/\log n \rceil)$ processors to retrieve the shortest path itself in logarithmic time, where k is the number of segments of that path. If the two query points are arbitrary rather than vertices, then one processor takes $O(\log n)$ time (instead of constant time) for finding the path length, while the complexity bounds for reporting an actual shortest path remain unchanged. A number of other related shortest paths problems are solved. The techniques we use involve a fast computation of staircase separators, and a scheme for partitioning the obstacles' boundaries in a way that ensures that the resulting path length matrices have a monotonicity property that is apparently absent before applying our partitioning scheme. Sequentially, the data structure can be built in $O(n^2)$ time.

1 Introduction

The problem of computing shortest paths that avoid obstacles is fundamental in computational geometry and has many applications. It has been studied in both sequential [8, 9, 11, 16, 17, 20–29, 31, 33, 38, 39] and parallel [12–15] settings, and using various metrics. The rectilinear version of the problem, which assumes that each path's constituent segments are parallel to the coordinate axes, is motivated by applications in areas such as wire layout, circuit design, plant and facility layout, urban transportation, and robot motion. There are many efficient sequential algorithms that compute rectilinear shortest paths avoiding

*Dept. of Computer Science, Purdue University, West Lafayette, IN 47907. This research was supported by the Office of Naval Research under Contracts N00014-84-K-0502 and N00014-86-K-0689, the Air Force Office of Scientific Research under Grant AFOSR-90-0107, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118.

†Dept. of Computer Science, Purdue University, West Lafayette, IN 47907.

different classes of polygonal obstacle sets [9, 11, 20, 22, 25, 26, 38, 39]. In this paper, we will present parallel techniques for solving several rectilinear shortest paths problems in the presence of rectangular obstacles.

The parallel computational model we use is the CREW-PRAM. Recall that this is the synchronous shared-memory model where concurrent reads are allowed, but no two processors can simultaneously attempt to write in the same memory location (even when they are trying to write the same thing).

We establish the following complexity bounds. Let P be a rectilinear convex polygon having $O(n)$ vertices and inside which lie n pairwise disjoint rectangular obstacles whose edges are parallel to the coordinate axes. We are interested in computing, in parallel, a data structure that supports queries about shortest rectilinear obstacle-avoiding paths in P. That is, a query specifies a source and a destination, and the data structure enables efficient processing of the query. We construct the data structure in $O(\log^2 n)$ time, with $O(n^2/\log^2 n)$ processors if all queries are such that the source and the destination are on the boundary of P, with $O(n^2/\log n)$ processors if the source is an obstacle vertex and the destination is on the boundary of P, and with $O(n^2)$ processors if both the source and destination are arbitrary points in the plane. The data structure we compute enables one processor to obtain the path length for any pair of query vertices (of obstacles or of P) in constant time, or $O(\lceil k/\log n \rceil)$ processors to retrieve the shortest path itself in logarithmic time, where k is the number of segments of that path. If the two query points are arbitrary rather than vertices, then one processor takes $O(\log n)$ time (instead of constant time) for finding the path length, while the complexity bounds for reporting an actual shortest path remain unchanged. We also solve the case when P is a convex N-gon with $n = o(N)$, in which case we are able to get an $O(N)$ rather than an $O(N^2)$ term in the work complexity by *implicitly* representing the $O(N^2)$ paths of interest, and the data structure for this implicit representation supports queries on lengths and paths within the same time and processor bounds as the data structure for the explicit representation. A number of other related shortest paths problems are solved. Sequentially, the data structure can easily be built in $O(n^2)$ time.

The techniques we develop involve a fast computation of staircase separators and a scheme for partitioning the obstacles' boundaries in a way that ensures that the resulting path length matrices have a monotonicity property that is apparently absent before applying our partitioning scheme. These techniques may be useful for other related problems. The

most general version of our algorithm uses a novel pipelining of the computation up and down the recursion tree, with $O(n)$ computational "flows" that originate from all nodes and proceed only to the nodes whose associated problem size is *larger* than that of the flow's origin.

De Rezende *et al.* [11] gave a sequential algorithm for computing rectilinear shortest paths avoiding a set of n rectangles between a *fixed* point s (the *source*) and arbitrary destination points in the plane. That is, the algorithm in [11] solves the *single source* case of the shortest path problem. In $O(n \log n)$ time, this algorithm constructs a data structure that can, in $O(\log n)$ time, answer a query that asks for the length of a rectilinear shortest path between the fixed source point s and an arbitrary destination point a. The data structure also enables the reporting of an actual rectilinear shortest path between s and a, in time proportional to the number of segments on the reported path. The method used in constructing the data structure of [11] is plane sweeping [32]. The queries we consider in this paper are more general than the ones in [11], because the data structure we build is for *all pairs* shortest paths between arbitrary points in the plane. Our algorithm is not a parallelized version of the algorithm in [11], and it indeed takes a very different approach to solve the problem. Recently, Guha and Stout [15] and, independently, ElGindy and Mitra [13] have given an $O(\log^3 n)$ time and $O(n^{1.5}/\log^2 n)$ processor algorithm for the special case where both the source and destination are fixed. Note that answering our queries using this approach would be inefficient, both in terms of the time and of the processor complexity.

The rest of the paper is organized as follows. Section 2 introduces some terminology and preliminary results. Section 3 gives one of the main ingredients we shall be using (the Staircase Separator Theorem). Section 4 proves some technical results that will be needed later in the "conquer" stages of our algorithms. Section 5 presents an algorithm which computes a data structure for an explicit representation for the lengths of the rectilinear shortest paths between the vertices of P for the case $|P| = O(n)$. Section 6 generalizes our solution to paths between arbitrary pairs of points (Subsection 6.3 is the most difficult part of the paper). Section 7 deals with the case $n = o(|P|)$. Section 8 extends the algorithms to computing the actual paths (rather than just their lengths). Section 9 sketches a sequential algorithm for building the data structure in $O(n^2)$ time. Section 10 concludes.

Throughout, all geometric objects (segments, polygons, paths, rectangles, etc.) are implicitly assumed to be rectilinear; that is, each of their constituent segments is parallel

to one of the two coordinate axes. From now on, all paths (shortest or otherwise) are assumed to be obstacle-avoiding. To avoid cluttering the exposition, we assume that no two distinct edges from P or R are collinear (the general case can be taken care of without much difficulty).

2 Preliminaries

A rectilinear convex polygon is a rectilinear simple polygon such that every line segment which joins two points of the polygon and is parallel to a coordinate axis is contained in the polygon.

The input polygon P is a convex polygon of N vertices. We use $Bound(P)$ to denote the boundary of P. Polygon P is specified by a circular sequence of vertices v_1, v_2, ..., v_N, as encountered by a counterclockwise walk along $Bound(P)$ starting at v_1. A circular ordering of the points on $Bound(P)$ is defined by the order in which they are encountered in the walk along $Bound(P)$ that follows the circular sequence of vertices of P. The boundary of P is said to be *clear* since it does not intersect the interior of any obstacle.

The set of rectangular obstacles is denoted by R. R is contained in P. The vertex set of R is denoted by V_R (hence $|V_R| = 4n$). We assume that V_R has already been sorted in $O(\log n)$ time using $O(n)$ processors [10].

We use $x(p)$ and $y(p)$ to denote the two coordinates of a point p. In the L_1 metric, the *distance* between two points p and q is $d(p,q) = |x(p) - x(q)| + |y(p) - y(q)|$. A segment with endpoints v and w is denoted by $\overline{vw}$ ($= \overline{wv}$). The *length* of a path C connecting two points is the sum of the lengths of its constituent segments. On the other hand, we use $|C|$ to denote the *size* of C, which is the number of segments of C (*not* its length).

A path is said to be *monotone* with respect to the x-axis (resp., y-axis) iff its intersection with every vertical (resp., horizontal) line is a contiguous portion of that line. A path is *convex* if it is monotone with respect to both the x-axis and the y-axis. A convex path has the shape of a staircase, and in fact we shall henceforth use the word "staircase" as a shorthand for "convex path". Note that a staircase from a point p to a point q is a shortest path between p and q since its length equals $d(p,q)$. Staircases can be *increasing* or *decreasing*, depending on whether they go up or down as we move along them from left to right. A staircase is *unbounded* if it starts and ends with a semi-infinite segment, i.e., a segment that extends to infinity on one side. A staircase is said to be *clear* if it does not

Figure 1: Illustrating $MAX_{NE}(R')$ and $MAX_{SW}(R')$.

intersect the interior of any obstacle.

A point p is strictly *below* (resp., to the *left* of) a point q iff $x(p) = x(q)$ and $y(p) < y(q)$ (resp., $y(p) = y(q)$ and $x(p) < x(q)$); we can equivalently say that q is strictly *above* (resp., to the *right* of) p. A rectangle r is *below* (resp., to the *left* of) an unbounded staircase S if no point of r is strictly above (resp., to the right of) a point of S; we can equivalently say that S is *above* (resp., to the *right* of) r.

For a subset R' of R, let S be a decreasing unbounded staircase that is above all rectangles in R'. Among all such staircases S, choose the *lowest-leftmost* one; that is, if S'' is the chosen one, then there is no unbounded decreasing staircase S' above R' with a point of S' strictly below or to the left of a point of S''. Denote such an S'' by $MAX_{NE}(R')$, where "N" is mnemonic for "North", and "E" is mnemonic for "East". Note that $MAX_{NE}(R')$ goes through all the maximal elements of $V_{R'}$ (see [32] for the definition of maximal elements of a point set). Using "S" and "W" as mnemonics respectively for "South" and "West", one can similarly define $MAX_{NW}(R')$, $MAX_{SE}(R')$, and $MAX_{SW}(R')$: $MAX_{NW}(R')$ is the lowest-rightmost increasing unbounded staircase above R', $MAX_{SE}(R')$ is the highest-leftmost increasing unbounded staircase below R', and $MAX_{SW}(R')$ is the highest-rightmost decreasing unbounded staircase below R'. See Figure 1.

The rectilinear convex hull of a set of objects in the plane, if it exists, is a (rectilinear) convex polygon that contains the set of objects and has minimum area [30]. In this paper, all convex hulls are rectilinear.

Given a subset R' of R, it is possible that the convex hull of R' does not exist (see [30] for example). This can happen in exactly one of two ways (but not both): (i) $MAX_{NE}(R')$ and $MAX_{SW}(R')$ intersect, or (ii) $MAX_{NW}(R')$ and $MAX_{SE}(R')$ intersect. In case (i) (resp., (ii))

Figure 2: Illustrating $Env(R')$ and the circular ordering on $Bound(Env(R'))$.

we define the convex connected region $Env(R')$ that contains R', called the *envelope* of R', as follows: consider the disconnected convex region of the plane that is below $MAX_{NE}(R')$ and $MAX_{NW}(R')$, and above $MAX_{SE}(R')$ and $MAX_{SW}(R')$, and let $Env(R')$ be the union of that region with the finite segments of $MAX_{NE}(R')$ (resp., $MAX_{NW}(R')$). Figure 2 (a) illustrates case (i), and Figure 2 (b) illustrates case (ii). Note that the definition of $Env(R')$ does not rule out that $Env(R')$ intersects the interior of an obstacle in $R - R'$; however, throughout the paper, we shall use the $Env(R')$ notation only in cases where $Env(R')$ does not intersect the interior of any obstacle in $R - R'$. Also note that if the convex hull of R' exists then it coincides with $Env(R')$ (see Figure 2 (c)). It is trivial to construct $Env(R')$ in $O(\log |R'|)$ time using $O(|R'|/\log |R'|)$ processors when $V_{R'}$ is already sorted, by using parallel prefix [18, 19] and parallel merging [35].

Let R' be a subset of R such that $Env(R')$ does not intersect the interior of any obstacle in $R - R'$. We now extend the circular ordering on the points of $Bound(Q)$ we defined earlier (where Q was a polygon) to the case when $Q = Env(R')$. We need to be able to say, for *any* three points p, p', p'' on $Bound(Q)$ (cf. Figure 2 (a)), that (for example) p' is *between* p and p'' *in the (extended) circular ordering* (i.e., starting at p and moving along the circular ordering we encounter p' before p''). For each $X \in \{NE, NW, SE, SW\}$, we define $MAX_X(Q)$ similarly to the way we defined $MAX_X(R')$. Observe that there is an obvious total ordering that one can define for the points of $MAX_X(Q)$ that are on the boundary of $Env(R')$ (i.e., $MAX_X(Q) \cap Bound(Q)$). The circular ordering we seek can then be viewed as the concatenation of these four total orderings. The concatenation may result in some points (from $MAX_{NE}(R')$ in case (i), and from $MAX_{NW}(R')$ in case (ii)) appearing more than once in the ordering, and we duplicate those points and treat them as

Figure 3: Illustrating $B(Q)$.

different points on $Bound(Q)$. More formally, the circular ordering is the circular version of the total order obtained as follows: start with the (totally ordered) points of $Bound(Q) \cap MAX_{NE}(Q)$, followed by those on $(Bound(Q) \cap MAX_{NW}(Q)) - MAX_{NE}(Q)$, followed by those on $(Bound(Q) \cap MAX_{SW}(Q)) - MAX_{NW}(Q)$, and followed by those on $(Bound(Q) \cap MAX_{SE}(Q)) - (MAX_{SW}(Q) \cup MAX_{NE}(Q))$.

Let Q be a convex connected region containing R', for a subset R' of R, such that Q does not intersect the interior of any obstacle in $R - R'$ (hence $Bound(Q)$ is clear). In particular, Q can be either $Env(R')$ or a convex polygon. In what follows, when we talk about "visibility", it is assumed that the obstacles as well as $Bound(Q)$ are opaque.

Definition 1 *Let $B(Q)$ be the set of points p on $Bound(Q)$ such that either (i) p is a vertex of Q, or (ii) p is horizontally or vertically visible from a vertex in $V_{R'}$ or from a vertex of Q (see Figure 3).*

That is, point $p \in Bound(Q)$ is in $B(Q)$ iff there is a vertex v of Q or of an obstacle contained in Q, such that segment $\overline{pv}$ is horizontal or vertical, and the interior of $\overline{pv}$ does not intersect $Bound(Q)$ or any obstacle. Obviously, $|B(Q)| = O(|Q|+|R'|)$. Using [4] and parallel merging [35], $B(Q)$ can be computed in $O(\log|Q| + \log|R'|)$ time and $O(|Q| + |R'|\log|R'|)$ work. We assume that $B(Q)$ is sorted according to the order in which its points are visited by a counterclockwise walk around $Bound(Q)$, starting at some vertex.

We shall repeatedly make use of Brent's theorem [7].

Theorem 1 (Brent) *Any synchronous parallel algorithm taking time T that consists of a total of W operations can be simulated by P processors in time $O((\mathsf{W}/\mathsf{P}) + \mathsf{T})$.*

There are actually two qualifications to the above Brent's theorem before one can apply it to a PRAM: (i) at the beginning of the i-th parallel step, we must be able to compute the amount of work W_i done by that step, in time $O(W_i/P)$ and with P processors, and (ii) we must know how to assign each processor to its task. Both qualifications (i) and (ii) to the theorem will be easily satisfied in our algorithms, therefore the main difficulty will be how to achieve W operations in time T.

Another result we shall be using deals with multiplying special kinds of matrices. All matrix multiplications are henceforth assumed to be in the $(\min, +)$ closed semi-ring, i.e., $(M' * M'')(i,j) = \min_k \{M'(i,k) + M''(k,j)\}$. If X, Y, and Z are finite sets of points in the plane, and if M_{XZ} (resp., M_{ZY}) denotes the matrix containing the lengths of the shortest paths from X to Z (resp., Z to Y), then it is not hard to see that the matrix $M_{XZ} * M_{ZY}$ contains the lengths of the shortest X-to-Y paths that are *constrained to go through* Z (i.e., they might not be best in absolute terms). Of course if for every path $\mathcal{P}$ from $p \in X$ to $q \in Y$ there exists a p-to-q path $\mathcal{P}'$ that goes through Z and is not longer than $\mathcal{P}$, then $(M_{XZ} * M_{ZY})(p,q)$ does contain the length of a shortest (unconstrained) p-to-q path.

A matrix M is said to be *Monge* [1] iff for any two successive rows i, $i+1$ and columns j, $j+1$ we have $M(i,j) + M(i+1,j+1) \leq M(i,j+1) + M(i+1,j)$. Now, consider two finite point sets X and Y, each totally ordered in some way (so we can talk about the predecessor and successor of a point in X or in Y), and such that that the rows (resp., columns) of the path lengths matrix M_{XY} are as in the ordering for X (resp., Y). Matrix M_{XY} is *Monge* iff for any two successive points p, p' in X and two successive points q, q' in Y we have $M_{XY}(p,q) + M_{XY}(p',q') \leq M_{XY}(p,q') + M_{XY}(p',q)$. Figure 4 gives examples for M_{XY}. Suppose that Q is a connected region whose boundary is clear and that X and Y are two finite point sets that are on two disjoint portions of the boundary of Q. In Figure 4 (a), Q is convex, and hence M_{XY} is Monge (assuming the points in X (resp., Y) are ordered as shown by the arrow). Figure 4 (b) shows an X and a Y for which M_{XY} is non-Monge (this figure also illustrates how length matrices that are non-Monge can arise in our problem). We shall later frequently make statements like "M_{XY} is Monge (or non-Monge)" without explicitly specifying what ordering we are assuming for the points in X and Y, when such an ordering is obvious from the context; for example, if X and Y are each a contiguous subset of the vertices of a convex polygon Q and are on two disjoint portions of *Bound*(Q) (as in Figure 4 (a)), then the implicit ordering assumed for X and Y is the obvious one for which M_{XY} is Monge (X in clockwise order along Q's boundary and Y in counterclockwise

Figure 4: Illustrating Monge and non-Monge matrices of path lengths.

order, or X in counterclockwise order and Y in clockwise order). The following lemma summarizes these easy observations.

Lemma 1 *Let CR be a convex connected region whose boundary is clear. Let X and Y be finite sets of points on the boundary of CR, such that the portion of that boundary spanned by X is disjoint from that spanned by Y (as in Figure 4 (a)). The matrix M_{XY} of path lengths between X and Y is Monge.*

The next lemma is frequently used later.

Lemma 2 *Let X and Y be two finite point sets that belong to two unbounded staircases S_X and (respectively) S_Y. Assume that S_X and S_Y are both clear. If X is completely on one side of S_Y, and Y is completely on one side of S_X, then M_{XY} is Monge.*

Proof. It is easy to see that the lemma's hypotheses imply the existence of a convex connected region CR having the properties stated in Lemma 1. $\square$

The following lemma is well known [3, 1].

Lemma 3 *Assume M_{XZ} and M_{ZY} are Monge, with $|X| = c_1|Z| \leq c_2|Y|$ for positive constants c_1 and c_2. Then $M_{XZ} * M_{ZY}$, which is also Monge, can be computed in $O(\log |Z|)$ time and $O(|X||Y|)$ work in the CREW-PRAM model.*

The next two lemmas are easy consequences of the previous one.

Lemma 4 *Let M_{XZ} and M_{ZY} be Monge, where $|X| \leq \alpha$, $|Y| \leq \beta$, and $|Z| \leq \gamma$, such that $\alpha = c_1\gamma \leq c_2\beta$ for positive constants c_1 and c_2. Then $M_{XZ} * M_{ZY}$ (which is also Monge) can be computed in $O(\log \gamma)$ time and $O(\alpha\beta)$ work in the CREW-PRAM model.*

Proof. "Pad" the matrices M_{XZ} and M_{ZY} with $+\infty$ entries so that they become $M_{X'Z}$ and $M_{ZY'}$, where $|X'| = \alpha$ and $|Y'| = \beta$. Apply Lemma 3 to multiply these padded matrices. The $M_{XZ} * M_{ZY}$ product is readily available from the $M_{X'Z} * M_{ZY'}$ product. □

Lemma 5 (Monge Multiply) *Let X, Y, and Z be finite point sets such that for any $p \in X$ and $q \in Y$, a shortest p-to-q path can be chosen to go through Z, where $|X| \leq \alpha$, $|Y| \leq \beta$, and $|Z| \leq \gamma$, such that $\alpha = c_1\gamma \leq c_2\beta$ for positive constants c_1 and c_2. Assume that X (resp., Y, Z) can be partitioned into a constant number of subsets X_i, $1 \leq i \leq l_X$ (resp., Y_j, Z_k, $1 \leq j \leq l_Y$, $1 \leq k \leq l_Z$) such that all $M_{X_iZ_k}$ and $M_{Z_kY_j}$ are Monge. Given M_{XZ} and M_{ZY}, the matrix M_{XY} can be computed in $O(\log\gamma)$ time and $O(\alpha\beta)$ work in the CREW-PRAM model.*

Proof. Trivial. □

3 Computing a Staircase Separator

This section establishes the following theorem:

Theorem 2 (Staircase Separator) *In $O(\log n)$ time and using $O(n)$ processors, it is possible to find an unbounded staircase, Sep, which partitions R into two subsets R_1, R_2 such that the following properties hold:*

1. *Sep does not intersect the interior of any obstacle in R.*
2. *Each of R_1 and R_2 contains no more than $7n/8$ rectangular obstacles.*
3. *Sep consists of $O(n)$ segments.*

Note: It is trivial to prove the *existence* of a *Sep*" for which $|R_1| = |R_2| = n/2$. The main contribution of this theorem is the parallel algorithm.

The rest of this section proves the Staircase Separator Theorem. We first introduce some terminology. For any point p, the *NorthWest* path of p (denoted by the shorthand $NW(p)$) is the path to infinity obtained by starting at p and going north until reaching an obstacle, at which point we go west along the obstacle's boundary until we clear the obstacle and are able to resume our trip north. One can in this way define an $XY(p)$ path and a $YX(p)$ path for any combination of $X \in \{N,S\}$ and $Y \in \{E,W\}$. An $XY(p)$ path starts at p and goes in the X direction whenever it can, and uses a "go in the Y direction" policy for getting around obstacles. A $YX(p)$ path is defined similarly. See 5 for example.

Figure 5: Illustrating $NE(p)$ and $WS(p)$.

To prove the theorem, it clearly suffices to find an unbounded staircase of size $O(n)$ that does not properly intersect any obstacle in R (it may run along an obstacle's boundary, however) and that has no less than $n/8$ obstacles on either side of it. The following lemma is one of the ingredients that will be used in computing such a staircase.

Lemma 6 (Path Tracing) *Given a point p not in the interior of any obstacle, an $XY(p)$ or a $YX(p)$ path can be computed in $O(\log n)$ time using $O(n)$ processors, where $X \in \{N, S\}$ and $Y \in \{E, W\}$.*

Proof. Without loss of generality (WLOG), we just show how to compute $NW(p)$ (the other $XY(p)$ and $YX(p)$ paths can be obtained similarly). The ingredients we need for this computation are the parallel trapezoidal decomposition method [4] and the Euler Tour technique for tree computation [36]. Let the bottom edge of each obstacle have a "parent" pointer to the left edge of the obstacle. Using the algorithm in [4] we obtain, for the upper-left vertex v of each obstacle, the trapezoidal segment above v (the trapezoidal segment is thus above the left edge containing v). The trapezoidal segment for point p is easy to find. These trapezoidal segments are the bottom edges of obstacles. (In the case where a trapezoidal segment does not exist, we assume that it is the "segment at infinity".) Then let p and the left edges of the obstacles each have a "parent" pointer to their respective trapezoidal segments. In this way, we create a forest whose nodes are left edges and bottom edges of obstacles, and point p. The roots of the trees in the forest are the nodes whose trapezoidal segment is at infinity. Using the Euler Tour technique for tree computation [36], we find the path from p to the root of the tree to which p belongs. The path so found is $NW(p)$. □

The algorithm for computing the desired staircase separator *Sep* is as follows: we first find a vertical line V such that there are as many vertices of R to its left as to its right. Let v be the number of obstacles in R that are properly intersected by V. If $v \geq n/4$ then we are essentially done: we find a point p on V such that half of the obstacles properly intersected by V are above it, and half of them below it. Assume that p is not in any obstacle (the algorithm can be easily modified for the case when p lies inside an obstacle). Then we take *Sep* to be the union of $NE(p)$ and $SW(p)$. So suppose, in what follows, that $v < n/4$. Find a horizontal line H such that there are as many vertices of R above it as below it. Let h be the number of obstacles in R properly intersected by H. If $h \geq n/4$ then we are done for the same reason as in the case where $v \geq n/4$. So suppose, in what follows, that $h < n/4$. Let p be the intersection of V and H, and assume that p is not in any obstacle (the algorithm can be easily modified for the case when p lies inside an obstacle).

Lines V and H together partition the plane into four quadrants which we call NE (NorthEast), NW, SE and SW. Let R_{NW} be the subset of R that lies only in the NW quadrant (hence no obstacle in R_{NW} properly intersects either V or H). Let R_{NE}, R_{SE}, and R_{SW} be defined analogously. Note that

$$|R_{NE}| + |R_{NW}| + |R_{SE}| + |R_{SW}| = n - v - h.$$

WLOG, assume that

$$|R_{NW}| = \max\{\ |R_{NE}|\ ,\ |R_{NW}|\ ,\ |R_{SE}|\ ,\ |R_{SW}|\ \}.$$

We now show that *Sep* can be taken to be the union of $NE(p)$ and $WS(p)$. Since such a *Sep* is obviously a staircase that consists of no more than $2n+2$ segments, does not properly intersect any obstacle, and separates R into two subsets, it suffices to prove that there are (i) at least $n/8$ obstacles above *Sep* and (ii) at least $n/8$ obstacles below *Sep*. Now, (i) is trivially true because, since each of h and v is less than $n/4$, we must have $|R_{NE}| + |R_{NW}| + |R_{SE}| + |R_{SW}| > n/2$, which implies $|R_{NW}| > n/8$. The proof of (ii) requires some work. Suppose to the contrary that there are fewer than $n/8$ obstacles below *Sep*. The staircase *Sep* partitions R_{NE} into two subsets: call them R'_{NE} and R''_{NE} (see Figure 6). Similarly, *Sep* partitions R_{SW} into two subsets: call them R'_{SW} and R''_{SW} (see Figure 6). WLOG, assume that $|R'_{NE}| \geq |R'_{SW}|$ (the other case is symmetrical). We obtain a contradiction to the definition of H, as follows. The number of vertices of R above H is $\geq 4|R_{NW}|+2h+4|R'_{NE}|+4|R''_{NE}| \geq 4|R_{NW}|+2h+4|R'_{NE}|$. The number of vertices of R below

Figure 6: Illustrating the algorithm for *Sep*.

H is $< 4(n/8) + 2h + 4|R'_{SW}|$ (where we used the assumption that there are fewer than $n/8$ obstacles below *Sep* and the fact that the number of obstacles that are simultaneously below both *Sep* and H is no more than the number of obstacles that are below *Sep*). Now, let us compare $4|R_{NW}| + 2h + 4|R'_{NE}|$ (which is less than or equal to the number of vertices of R above H) with $4(n/8) + 2h + 4|R'_{SW}|$ (which is strictly larger than the number of vertices of R below H). Since $|R_{NW}| > n/8$ and $|R'_{NE}| \geq |R'_{SW}|$, we have

$$4|R_{NW}| + 2h + 4|R'_{NE}| > 4(n/8) + 2h + 4|R'_{SW}|.$$

It follows that the number of vertices of R below H is smaller than the number of vertices of R above H. This contradicts the definition of H, and completes the proof of the Staircase Separator Theorem.

4 Other Building Blocks

This section introduces further technical results that will later be used. In what follows, Q is a convex connected region containing a subset R' of R such that either (i) Q is a convex polygon with $O(|R'|)$ vertices, or (ii) $Q = Env(R')$. The lemmas in this section assume that Q does not intersect the interior of any obstacle in $R - R'$. Note that the boundary of Q is clear. For such a Q, we define arrays *Horiz* and *Vert* (of size $|B(Q)|$ each) as follows. Let p, q be a pair of adjacent points in $B(Q)$; that is, $\overline{pq}$ is on $Bound(Q)$ and p, q are the only points of $B(Q)$ that are on $\overline{pq}$. Then $Horiz(\overline{pq})$ (resp., $Vert(\overline{pq})$) is the portion of $Bound(Q)$ $- \overline{pq}$ that is horizontally (resp., vertically) visible from $\overline{pq}$; that is, either $Horiz(\overline{pq})$ (resp., $Vert(\overline{pq})$) is empty, or for each point $a \in Horiz(\overline{pq})$ (resp., $a \in Vert(\overline{pq})$) there is a point b $\in \overline{pq}$ such that a is horizontally (resp., vertically) visible from b. In Figure 7, $Vert(\overline{pq}) =$

Figure 7: Illustrating array *Vert*.

$\overline{p'q'}$, and $Vert(\overline{qr})$ is empty. The procedures that later use these lemmas will always make sure that the *Horiz* and *Vert* arrays are available (it is in fact quite easy to compute these arrays, by using parallel prefix [18, 19]).

When computing the shortest paths between pairs of *vertices* of Q, we shall also concern ourselves with the nonvertex points in $B(Q)$. The reason we do this is that (as will become apparent later) it is easier to solve the more general problem of computing the $B(Q)$-to-$B(Q)$ paths.

Notation 1 *We use* D_Q *to denote the* $|B(Q)| \times |B(Q)|$ *matrix containing the lengths of shortest paths between all pairs of points in* $B(Q)$.

Lemma 7 (Discretization) *Given the matrix* D_Q *and arrays Horiz and Vert, the length of a shortest path between any pair of points on Bound*(Q) *can be found in* $O(\log |B(Q)|)$ *time using one processor.*

Proof. Let b_1 and b_2 be two points on *Bound*(Q). Let v (resp., w) be the first point of $B(Q)$ encountered by a clockwise (resp., counterclockwise) walk from b_1 along *Bound*(Q). If $b_1 \in B(Q)$, then $b_1 = v = w$. Let points v' and w' be similarly defined for b_2. WLOG, assume that both b_1 and b_2 are not in $B(Q)$. The $O(\log |B(Q)|)$ time is needed only for finding $\overline{vw}$ and $\overline{v'w'}$. If $\overline{vw}$ is contained in $Horiz(\overline{v'w'})$ or in $Vert(\overline{v'w'})$, or if $\overline{v'w'}$ is contained in $Horiz(\overline{vw})$ or in $Vert(\overline{vw})$, then the b_1-to-b_2 path length is simply $d(b_1, b_2)$. Otherwise the path length we seek is one of the four following quantities: (i) $d(b_1, v) + D_Q(v, v') + d(v', b_2)$, (ii) $d(b_1, v) + D_Q(v, w') + d(w', b_2)$, (iii) $d(b_1, w) + D_Q(w, v') + d(v', b_2)$, and (iv) $d(b_1, w) + D_Q(w, w') + d(w', b_2)$. This can be proved by contradiction: assuming that none of (i)–(iv)

Figure 8: Illustrating Lemma 8.

is the length we seek leads to a contradiction with the definition of one of $\{v, w\}$ or $\{v', w'\}$. □

To avoid introducing new notation, we shall from now on use $Env(X)$ even when X consists of arbitrary objects (not just rectangular obstacles). The definition we gave earlier for the case $X = R$ extends to other objects in a natural way. In particular, X can now be a collection of polygons, staircases, etc.

Lemma 8 (Staircase Extension) *Let C be a bounded staircase originating on $Bound(Q)$ such that (i) C is a contiguous portion of the boundary of $Q' = Env(Q \cup C)$, and (ii) Q' intersects the interior of an obstacle only if the obstacle is contained in Q. Let C' (resp., B') be $B(Q') \cap C$ (resp., $B(Q') \cap Bound(Q)$). Then given the matrix D_Q, we can obtain the matrix of the B'-to-C' path lengths in $O(\log m)$ time and $O(m^2)$ work, where $m = |C| + |B(Q)|$.*

Proof. WLOG, we assume that C starts at the highest edge of Q and is decreasing (Figure 8). Let $Cross$ be the set of points on $Bound(Q) - Bound(Q')$ that either are in $B(Q)$ or are horizontal or vertical projections of the vertices of C. We partition $Cross$ into two subsets: $Cross_1$ which contains those points of $Cross$ on $MAX_{NE}(Q)$, and $Cross_2 = Cross - Cross_1$ (see Figure 8). The matrix M of the B'-to-$Cross$ path lengths can be obtained from D_Q within the desired complexity bounds, by using the Discretization Lemma (Lemma 7), and similarly for the matrix M' of the $Cross_2$-to-$Cross_1$ path lengths. The matrix M_1 of the $Cross_1$-to-C' path lengths is trivially available (each v-to-w path length in it is simply $d(v, w)$). The lengths of shortest paths between $Cross_2$ and the portion of C' that is above $Cross_1$ can be obtained by multiplying M' with M_1; since both M' and M_1 are Monge

(by Lemma 1), they can be multiplied within the desired complexity bounds (by using the Monge Multiply Lemma (Lemma 5)). The lengths of shortest paths between $Cross_2$ and the portion of C' that is not above $Cross_1$ are trivial to obtain (they are described by the function $d(\cdot\ ,\ \cdot)$). Hence we now have the matrix M^* of the lengths of the $Cross$-to-C' paths. To obtain the lengths of the B'-to-C' paths, we use the Monge Multiply Lemma on length matrices M and M^*, with B' playing the role of X, C' playing the role of Y, and $Cross$ playing the role of Z. □

Lemma 9 *Let Sep' be the staircase obtained by applying the Staircase Separator Theorem (Theorem 2) to R', and let R'_1 and R'_2 be the two subsets of R' on either side of Sep'. Then both $Bound(Env(R'_1))$ and $Bound(Env(R'_2))$ are clear.*

Proof. This follows from the facts that Sep' is a staircase that does not properly intersect the obstacles in R', that $Env(R'_1)$ and $Env(R'_2)$ are both contained in Q, and that Q does not intersect the interior of any obstacle in $R - R'$. □

Lemma 10 (Containment) *Let points q_1 and q_2 belong to Q and let $\mathcal{P}$ be a path between q_1 and q_2. Then there exists a path $\mathcal{P}'$ between q_1 and q_2 which does not go outside of Q and is not longer than $\mathcal{P}$.*

Proof. Since Q is a convex connected region whose boundary is clear, any portion of $\mathcal{P}$ that goes outside Q can be replaced by going along the boundary of Q. The length of the path $\mathcal{P}'$ obtained from the replacement is not longer than that of $\mathcal{P}$ because of the convexity of Q. □

Lemma 11 (Single Intersection) *If a shortest path between points p and q intersects a clear staircase S', then there exists a shortest path between p and q whose intersection with S' is one connected component.*

Proof. This is an immediate consequence of the fact that for any two points s_1 and s_2 of S', a shortest path between them is the path along S'. □

5 Computing the Lengths Matrix D_P When $|P| = O(|R|)$

Recall that the input polygon P is convex and contains all the obstacles in R, and that D_P is the matrix of the $B(P)$-to-$B(P)$ shortest path lengths. In this section, we assume that $|P| = N \leq c|R|$ for some positive constant c, and we only concern ourselves with computing

D_P. It suffices to give an algorithm for the case where the input consists of only R and where we wish to compute the lengths of paths between pairs of points in $B(Q)$ where $Q = Env(R)$. This is enough because if the input includes both P and R, then we first compute D_Q and then easily obtain D_P from it with a constant number of applications of the Staircase Extension Lemma (Lemma 8).

The algorithm takes as input the set R of n rectangular obstacles, and computes the $|B(Q)| \times |B(Q)|$ matrix D_Q, where $Q = Env(R)$. It does so by first finding a staircase separator *Sep* that partitions R into two subsets R_1 and R_2. Then it recursively solves, in parallel, the subproblems for R_1 and R_2, respectively, obtaining two matrices D_{Q_1} and D_{Q_2}, where $Q_1 = Env(R_1)$ and $Q_2 = Env(R_2)$. Finally it obtains matrix D_Q from matrices D_{Q_1} and D_{Q_2}.

We use the Staircase Separator Theorem (Theorem 2) to find *Sep*. Computing Q_1 and Q_2 is trivial. Because of Lemma 9 and the Containment Lemma (Lemma 10), the two matrices returned by the two recursive calls contain, respectively, the lengths of the $B(Q_1)$-to-$B(Q_1)$ paths and the $B(Q_2)$-to-$B(Q_2)$ paths (i.e., they are indeed D_{Q_1} and D_{Q_2}). Thus the main difficulty is how to efficiently obtain D_Q from D_{Q_1} and D_{Q_2}.

Let $T(n)$ and $W(n)$ respectively denote the time and work complexities of the algorithm. Then to show that $T(n) = O(\log^2 n)$ and $W(n) = O(n^2)$, it suffices to prove Theorem 3 below. This would be enough because we would then have:

$$\begin{aligned} T(n) &\leq T(7n/8) + c_1(\log n) \\ W(n) &\leq W(|R_1|) + W(|R_2|) + c_2(n^2) \end{aligned}$$

with the boundary conditions $T(1) = c_3$ and $W(1) = c_4$, where the c_i's are positive constants, $|R_1| + |R_2| = n$, $n/8 \leq |R_1|, |R_2| \leq 7n/8$. Brent's theorem [7] would then imply a processor complexity of $O(n^2/\log^2 n)$.

Theorem 3 *The matrix D_Q can be computed from D_{Q_1} and D_{Q_2} in $O(\log n)$ time and $O(n^2)$ work.*

Proof. Let Q_{left} (resp., Q_{right}) be the portion of Q on the left (resp., right) side of *Sep* (see Figure 9). (Note that Q_{left} and Q_{right} both include the portion of *Sep* that is in Q.) Since Q_1 is contained in Q_{left} and the matrix D_{Q_1} is known, we can apply the Discretization Lemma (Lemma 7) and the Staircase Extension Lemma (Lemma 8) a constant number of times to obtain the matrix $D_{Q_{left}}$. The matrix $D_{Q_{right}}$ is obtained

Figure 9: Illustrating the proof of Theorem 3.

similarly. Let *Left* (resp., *Right*) be the subset of $B(Q)$ that is in Q_{left} (resp., Q_{right}), and let *Middle* be the subset of $B(Q_{left}) \cup B(Q_{right})$ that lies on *Sep*. From matrix $D_{Q_{left}}$ (resp., $D_{Q_{right}}$), using the Discretization Lemma, we can obtain the matrix M_{left} (resp., M_{right}) of the lengths of shortest paths between *Left* (resp., *Right*) and *Middle*. The Single Intersection Lemma (Lemma 11) implies that the problem of computing D_Q is essentially that of multiplying M_{left} and M_{right}. By Lemma 1, these two matrices are Monge. Hence by using the Monge Multiply Lemma (Lemma 5), these two matrices can be multiplied within the desired bounds. The correctness of the computation of D_Q easily follows from the fact that for any points p, q, where $p \in Left$ and $q \in Right$, there exists a p-to-q shortest path that goes through a point in *Middle*. □

6 Path Lengths between Arbitrary Points

We extend the techniques of the previous sections to computing the lengths of shortest paths between arbitrary query points. The query time is logarithmic using one processor. We first consider the structure for the $B(P)$-to-V_R paths and construct it using an $O(\log^2 n)$ time algorithm with $O(n^2/\log n)$ processors. We then consider the structure for the V_R-to-V_R paths and construct it using an $O(\log^2 n)$ time algorithm with $O(n^2)$ processors. Finally, we show that even with arbitrary query points we can use essentially the same structure as in the V_R-to-V_R case. The first subsection gives some observations that are crucial in all of the above cases.

Figure 10: Illustrating U, U', W, and W' at a node v of T.

6.1 Some Useful Observations

Let T be the recursion tree for the algorithm in Section 5; that is, the root of T corresponds to the "top-level" recursive call (the one associated with R), the children of the root correspond to the recursive calls for R_1 and R_2, and so on. It is easy to modify that algorithm so that the information (path length matrices, separators, etc.) produced by each recursive call remains stored in T even after that call returns. We assume that this modification has already been done, so that each node v of T stores the obstacle set $R_v \subseteq R$ associated with v, as well as $Q_v = Env(R_v)$, the staircase Sep_v partitioning R_v (WLOG, assume Sep_v is increasing), and the following matrices in addition to matrix D_{Q_v}. Let $LeftR_v$ (resp., $RightR_v$) be the subset of R_v to the left (resp., right) of Sep_v. Let $Left\text{-}Sep_v$ (resp., $Right\text{-}Sep_v$) be the bounded staircase consisting of the portion of $MAX_{SE}(Env(LeftR_v))$ (resp., $MAX_{NW}(Env(RightR_v))$) that is in the interior of Q_v. Let U_v consist of the subset of $B(Env(LeftR_v \cup Left\text{-}Sep_v))$ that is on $Left\text{-}Sep_v$. Let U'_v be the subset of $B(Env(LeftR_v))$ that is in the interior of Q_v and is not on $Left\text{-}Sep_v$ (see Figure 10). Let W_v be the subset of $B(Env(RightR_v \cup Right\text{-}Sep_v))$ on $Right\text{-}Sep_v$, and let W'_v be the subset of $B(Env(RightR_v))$ that is in the interior of Q_v and is not on $Right\text{-}Sep_v$ (see Figure 10). The additional matrices we store at node v are (i) $M_{v,U}$ for the lengths of the U_v-to-$B(Q_v)$ paths, (ii) $M_{v,U'}$ for the lengths of the U'_v-to-$B(Env(LeftR_v \cup Left\text{-}Sep_v))$ paths, (iii) $M_{v,W}$ (with obvious meaning), and (iv) $M_{v,W'}$. The reader may observe that the above four matrices were not explicitly computed by the algorithm in Section 5, but it is easy to modify that algorithm so that it does compute them, using the Discretization Lemma (Lemma 7) and the Staircase Extension Lemma (Lemma 8).

The storage space taken by T and all the information associated with its nodes obeys the same recurrence as for the work complexity, and hence is $O(n^2)$.

For convenience, we now introduce a notation $Chain(\cdot)$ such that, if X is a finite set of points that were obtained from some contiguous portion of a staircase, then $Chain(X)$ is that contiguous portion of the staircase; usually the context makes it clear which contiguous portion of the staircase is meant—we shall typically use $Chain(X)$ for $X \in \{U_v, U'_v, W_v, W'_v\}$. For example, $Chain(U_v) = Left\text{-}Sep_v$, and $Chain(U'_v) =$ the portion of $Bound(Env(LeftR_v))$ that is in the interior of Q_v and is not on $Left\text{-}Sep_v$. Observe that staircases $Chain(U_v)$ and $Chain(W_v)$ both divide Q_v into two halves, each of which is a convex connected region, whereas staircases $Chain(U'_v)$ and $Chain(W'_v)$ respectively cut $Env(LeftR_v \cup Left\text{-}Sep_v)$ and $Env(RightR_v \cup Right\text{-}Sep_v)$ into two halves, each of which is also a convex connected region.

Each obstacle vertex $p \in V_R$ occurs on at least one of the U_v, U'_v, W_v, W'_v lists, for some $v \in T$. Therefore to compute the V_R-to-$B(P)$ path lengths, it suffices to compute, for all $v \in T$ and $X \in \{U, U', W, W'\}$, the X_v-to-$B(P)$ path lengths. The reader may wonder why we have partitioned the points in $B(Env(LeftR_v)) - Bound(Q_v)$ into two subsets U_v and U'_v: the reason is that it will enable the use of the Monge Multiply Lemma (Lemma 5), by making the path length matrices Monge, something which would not have been true otherwise (this will become clearer in the proofs of the lemmas below).

We henceforth assume that a pre-processing stage has explicitly computed, for each $p \in V_R$, the eight paths $X(p)$ for all $X \in \{NE, NW, SE, SW, EN, ES, WN, WS\}$ (the definitions of these paths were given in Section 3; see Figure 5 for example). This is done by first computing the forest that implicitly describes all the $NE(p)$'s (call it the "NE forest") in $O(\log n)$ time with $O(n)$ processors, as in the proof of the Path Tracing Lemma (Lemma 6). Then we extract from that NE forest an explicit description of $NE(p)$, for each $p \in V_R$. This extraction is easily done in $O(\log n)$ time and $O(n^2)$ work, by making a copy of the tree that contains p for each $p \in V_R$ and obtaining $NE(p)$ from that copy using standard parallel tree computation methods [36]. Given points p and q, where $p \in V_R$ and q is arbitrary, determining whether $NE(p)$ goes above or below q can be done in logarithmic time using one processor (by a binary search on $NE(p)$). The same holds for the other 7 forests that describe the other 7 kinds of paths. We can speak of the *segments associated with a forest* (say, the NE forest): these are the segments that lie on $NE(p)$ for some $p \in V_R$. There are clearly $O(n)$ such segments associated with each of the 8 forests. In fact, all of the chains associated with the recursion tree's nodes (i.e., the chains for $\{U_v, U'_v, W_v, W'_v\}$) use only

segments associated with the eight forests. We pre-process the segments associated with these 8 forests in the following way: for each such forest (say, the NE one), we compute an *indicator* matrix I_{NE} of size $O(n) \times O(n)$ which is defined as follows. For each $p \in V_R$ and each segment s associated with the 8 forests, $I_{NE}(p,s) = s'$, where s' is the segment of $NE(p)$ that intersects the infinite line l_s containing s. These eight indicator matrices are easily computed in $O(\log n)$ time and using a quadratic amount of work. It is easily seen that these indicator matrices enable us to determine, for any point $p \in V_R$ and any staircase C which uses only segments associated with the 8 forests, whether, for example, $NE(p)$ intersects C, and to find a point on that intersection, in $O(\log |C|)$ time and $O(|C|)$ work. This last observation is used implicitly in the proof of the Bridging Lemma (Lemma 14). The next two lemmas are also needed for proving the Bridging Lemma.

Definition 2 *Two staircases $\mathcal{P}$ and $\mathcal{P}'$ are said to* cross once *iff (i) their intersection is not empty, (ii) each staircase has at least one point that is strictly to the left of the other staircase and one point that is strictly to its right, and (iii) for either staircase, the portion of that staircase that is on or to the left (resp., right) of the other staircase consists of one connected component. We adopt the convention that the* crossing point *between two such staircases is one that belongs to their intersection and partitions them into pieces that do not satisfy (ii) (if many such points can be so chosen, we choose the one with, say, the smallest x coordinate).*

Intuitively, "crossing once" means a staircase switching from being strictly on one side of the other staircase to being strictly on the other side of it, exactly one time. For example, two unbounded increasing staircases $\mathcal{P}$ and $\mathcal{P}'$ such that no point of $\mathcal{P}$ is strictly above $\mathcal{P}'$ cannot be said to cross once *even if their intersection is non-empty.*

Lemma 12 *Let C be a clear staircase. For any $p \in V_R$ and any $X \in \{NE, NW, SE, SW, EN, ES, WN, WS\}$, $X(p)$ crosses C at most once.*

Proof. If one of C and $X(p)$ is increasing and the other decreasing, then the lemma trivially holds. So suppose that both C and $X(p)$ are increasing (the proof is similar if they are both decreasing). To prove that C and $X(p)$ cross at most once, first observe that one of the two classes of segments of $X(p)$ (horizontal or vertical) consists of segments that coincide with obstacle boundaries. WLOG, assume the horizontal segments of $X(p)$ all coincide with obstacle boundaries. In order for C and $X(p)$ to cross more than once, at least one

vertical segment of C would have to properly intersect one of the horizontal obstacle edges along which runs one of $X(p)$'s horizontal segments. This would imply that C penetrates the interior of an obstacle, contradicting the hypothesis that C is clear. $\square$

Lemma 13 *Let v be a node of T and X be any of $\{U, U', W, W'\}$. For a point $p \in X_v$ and a point q not in the interior of Q_v, there exists a shortest p-to-q path that goes through a point of $B(Q_v)$.*

Proof. Let $\mathcal{P}$ be a shortest p-to-q path. Since q is not in the interior of Q_v, $\mathcal{P}$ must intersect $Bound(Q_v)$ before reaching p. By the Containment Lemma (Lemma 10), $\mathcal{P}$ can be chosen so that it enters Q_v only once, say, $\mathcal{P}$ intersects $Bound(Q_v)$ in between two adjacent points $b_1, b_2 \in B(Q_v)$. (Note that $\overline{b_1 b_2}$ is on $Bound(Q_v)$ and no other point of $B(Q_v)$ is on $\overline{b_1 b_2}$.) WLOG, assume $\overline{b_1 b_2}$ is vertical and the interior of Q_v is to its left. Imagine shooting leftward horizontal rays from all the points of $\overline{b_1 b_2}$, and let *Region* be the region illuminated by these rays, assuming that obstacles as well as $Bound(Q_v)$ are opaque. Point p cannot lie in the interior of *Region*, since otherwise b_1 and b_2 would not be adjacent in $B(Q_v)$ and would be separated in $B(Q_v)$ by the horizontal projection of p on $\overline{b_1 b_2}$. This means that $\mathcal{P}$ has to intersect one of the two rays from b_1 and (respectively) b_2, and hence can be deformed so that it goes through either b_1 (if it intersects the ray of b_1) or b_2. $\square$

Lemma 14 (Bridging) *Let X and Y be any of $\{U, U', W, W'\}$. Let v and w be two nodes of T such that $|R_v| \leq c|R_w|$ for some positive constant c and $Chain(Y_w)$ does not intersect the interior of Q_v. If, in addition to the information stored in T, we are given the lengths of the Y_w-to-$B(Q_v)$ paths, then we can compute, in $O(\log(|R_v|))$ time and $O(|R_v||R_w|)$ work, the lengths matrix of the shortest X_v-to-Y_w paths.*

Proof. We begin with the case $X_v = U_v$ or W_v; WLOG, assume $X_v = U_v$. Note that $Chain(X_v)$ partitions Q_v into two halves such that each half of Q_v is convex and connected.

Let p, p' be the endpoints of $Chain(X_v)$, and q, q' be the endpoints of $Chain(Y_w)$. WLOG, assume that $Chain(Y_w)$ is increasing, that q' is the lower-left endpoint of $Chain(Y_w)$, and that q is the upper-right endpoint of $Chain(Y_w)$. Now, augment $Chain(Y_w)$ by adding to it $NE(q)$ and $SW(q')$, thus obtaining an unbounded staircase $Chain'(Y_w)$. We distinguish two cases, depending on whether $Chain'(Y_w)$ intersects the interior of Q_v or not. Testing whether such an intersection occurs is easy to do, by using the indicator matrices.

Figure 11: Illustrating the proof of Lemma 14.

The first case, when $Chain'(Y_w)$ does not intersects the interior of Q_v, is handled as follows. WLOG, assume that Q_v is below $Chain'(Y_w)$. Let l, r, t, and b be respectively a leftmost, rightmost, top, and bottom vertex of Q_v (there are at most two candidates for each, and we choose one of these two arbitrarily). The idea is to use the Monge Multiply Lemma (Lemma 5), with $B(Q_v)$ playing the role of Z in that lemma, X_v playing the role of X in that lemma, and Y_w playing the role of Y in that lemma. (Note that by Lemma 13, the X_v-to-Y_w paths can be chosen to go through $B(Q_v)$.) But in order to be able to use that lemma, we need to judiciously partition each of $B(Q_v)$ and Y_w into a constant number of pieces (X_v will not need to be partitioned). The partitioning of $B(Q_v)$ is quite simple: the points determining the partition are l, r, t, b, p, and p' (see Figure 11); hence $B(Q_v)$ gets partitioned into at most six pieces—fewer if the six points determining the partition are not distinct. Note that the path lengths matrix between X_v and any of these six pieces is Monge (by Lemma 1), thus satisfying one of the requirements for the Monge Multiply Lemma. To satisfy the other requirement, however, we must partition Y_w with great care, in such a way that the path lengths matrix between each piece of Y_w and each piece of $B(Q_v)$ is indeed Monge. This partitioning of Y_w is induced by a partitioning of $Chain(Y_w)$ into at most seven pieces, according to the following (at most six) points: the points at which $Chain(Y_w)$ crosses each of $NE(r)$, $NE(t)$, $NW(t)$, $NW(l)$, $SW(l)$, and $SW(b)$ (see Figure 11). (Note that $Chain(Y_w)$ can cross each of $NE(r)$, $NE(t)$, $NW(t)$, $NW(l)$, $SW(l)$, and $SW(b)$ at most once, by Lemma 12.) Finding these six points is easy to do by using the indicator matrices. It is not hard to see that this is a suitable partition of Y_w, by Lemma 2.

The second case, when $Chain'(Y_w)$ intersects the interior of Q_v, is handled as follows.

Figure 12: Illustrating Lemma 15.

By Lemma 12, $Chain'(Y_w)$ can cross $Chain(X_v)$ at most once and $Bound(Q_v)$ at most twice. The crossing point between $Chain(X_v)$ and $Chain'(Y_w)$ (if one exists), as well as the (at most) two crossing points of $Chain'(Y_w)$ with the boundary of Q_v, can easily be computed by using the indicator matrices. $Chain'(Y_w)$ defines two independent subproblems, one on each side of it; they are independent because of the Containment Lemma (Lemma 10). We solve each of these two subproblems separately, similarly to the way we solved the first case.

We now turn our attention to the case $X_v = U'_v$ or W'_v; WLOG, assume $X_v = U'_v$. Suppose that we have computed the lengths of the U_v-to-Y_w paths using the algorithm in the previous paragraphs (hence the lengths of the Y_w-to-$B(Env(LeftR_v \cup Left\text{-}Sep_v))$ paths are known). Then essentially the same algorithm as for the case $X_v = U_v$ works except that $Env(LeftR_v \cup Left\text{-}Sep_v)$ now plays the role of Q_v and U'_v plays the role of U_v (Y_w being the same). □

Lemma 15 *Let w be an ancestor of v in T. Let X be any of $\{U, U', W, W'\}$. If, in addition to the information stored in T, we are given the lengths of the $B(Q_v)$-to-$B(Q_w)$ paths, then we can compute, in $O(\log(|R_v|))$ time and $O(|R_v||R_w|)$ work, the lengths matrix of the shortest X_v-to-$B(Q_w)$ paths.*

Proof. If $w = v$, then the computation is trivial. Otherwise, Q_w properly contains Q_v (see Figure 12). Hence $Bound(Q_w)$ does not intersect the interior of Q_v. Partition $Bound(Q_w)$ into four staircases, in the obvious way, and for each such staircase C use the same proof as in the Bridging Lemma (Lemma 14), with $B(Q_w) \cap C$ playing the role of Y_w. □

Lemma 16 *For each v in T and all $X, Y \in \{U, U', W, W'\}$, the lengths matrix of the X_v-to-Y_v paths can be computed in $O(\log |R_v|)$ time and $O(|R_v|^2)$ work (see Figure 10).*

Proof. Similar to that of the Bridging Lemma (Lemma 14) and omitted. □

The observations presented in this subsection will be used in what follows.

6.2 The $B(P)$-to-V_R Path Lengths

We begin with the case $P = Env(R)$. First, we construct the recursion tree T and all its associated information, as explained in the previous subsection. Let *root* be the root of T (hence $Q_{root} = Env(R)$). We would like to compute, for each node $v \in T$, the four matrices containing the X_v-to-$B(Q_{root})$ path lengths, for each $X \in \{U, U', W, W'\}$. We do this from the root down, one level at a time. At *root*, we use Lemma 15 to do this in $O(|R_{root}|^2)$ work (the condition for the lemma is trivially satisfied there, since we are using it with $root = v = w$). Having done this for *root* makes the application of Lemma 15 at each child v of *root* possible (with $w = root$), which takes $O(|R_{root}||R_v|)$ work for each such v. This in turn makes the application of the lemma at each grandchild v of the root possible, etc. We proceed in this way from the root down, one level at a time, until we reach the leaf level. Let the height of T be $height(T)$. The time for this is clearly $O(\log|R_{root}| * height(T))$ and the work is $O(|R_{root}|\sum_{v \in T}|R_v|)$. This implies an $O(\log^2 n)$ time and $O(n^2 \log n)$ work complexities (where the fact that $\sum_{v \in T}|R_v| = O(n \log n)$ was used). By Brent's theorem, the processor complexity is $O(n^2/\log n)$. The case where P properly contains $Env(R)$ is easily handled by the method for the above case, in conjunction with that of Section 5.

6.3 The V_R-to-V_R Path Lengths

First we do the following pre-processing. In parallel for each $w \in T$, we compute the lengths of the X_v-to-$B(Q_w)$ paths and the X_v-to-Y_w paths for all descendants v of w, and all $X, Y \in \{U, U', W, W'\}$. These two computations are trivial to do if $v = w$ (in the first case the information is already stored in T, in the second case we can use Lemma 16). So suppose $v \neq w$, i.e., v is a proper descendant of w. Then the computation of the X_v-to-$B(Q_w)$ path lengths is done exactly as in the previous subsection (with w now playing the role of *root*), resulting in $O(\log^2 n)$ time and $O(|R_w|^2 \log |R_w|)$ work for this particular w. This also gives us some but not all of the desired X_v-to-Y_w path lengths; for example, if u is the child of w whose Q_u contains X_v, and if U'_w is on $Bound(Q_u)$, then we already know the X_v-to-U'_w path lengths but not the X_v-to-W'_w path lengths—these must still be computed. We compute the remaining X_v-to-Y_w path lengths also in a top-down manner, in parallel for all w, from w down, by using repeatedly the Bridging Lemma (Lemma 14) at each level

of the downward trip from w; the lemma's hypothesis is satisfied, i.e., we do know the Y_w-to-$B(Q_v)$ path lengths, because they would already have been computed earlier by w's top-down computation. This too takes $O(\log^2 n)$ time and $O(|R_w|^2 \log |R_w|)$ work. Summed over all such w, the total work for the pre-processing is $O(\log n \sum_{w\in T} |R_w|^2) = O(n^2 \log n)$.

Since we already computed, in the previous subsection, the lengths of the paths having an endpoint in $B(Env(R))$, it suffices to compute the lengths of paths having both endpoints in $V_R - B(Env(R))$. Each vertex in $V_R - B(Env(R))$ appears on some X_v, $v \in T$, $X \in \{U, U', W, W'\}$. Therefore it suffices to compute the lengths of the X_v-to-Y_w paths for all $v, w \in T$ and $X, Y \in \{U, U', W, W'\}$. This is done in the rest of this subsection.

Before going into the details, we point out the main reason behind the elaborate constructions that are about to follow: unless great caution is exercised, when computing the X_v-to-Y_w path lengths for a particular v, w pair, the associated Monge matrix multiplication might not satisfy the size requirements of the Monge Multiply Lemma (Lemma 5); that is, the required relations between α, β, and γ of that lemma might be violated. This is the main reason for the condition "$|R_v| \leq |R_w|$" that is about to play such an important role in the concept of "flow" that is given next.

For nodes $v, w \in T$, let the *tree distance* between v and w, denoted by $l(v, w)$, be the number of edges on the v-to-w path in the undirected version of T. Clearly, $l(v, v) = 0$. The computation for the V_R-to-V_R path lengths proceeds in $2 * height(T)$ stages, each of which takes $O(\log n)$ time. Whereas the approach in the previous subsection was a "top-down flow" from the root of T, repeatedly making use of Lemma 15, here the flow is from each v to the w's that have $|R_w| \geq |R_v|$, in the order of their tree distance from v. The flows for all v's start at the same time. Thus, if $|R_v| \leq |R_w|$, then the flow for v reaches w at stage $l(v, w)$ (which is at most $2 * height(T)$). When the flow for v reaches w, it computes the desired information between v and w, possibly using the Monge Multiply Lemma (Lemma 5) and the Bridging Lemma (this information consists of more than the X_v-to-Y_w path lengths—more on this later). Observe that for any pair $v, w \in T$, the flow of one of these two nodes eventually reaches the other, so that all the X_v-to-Y_w path lengths eventually gets computed. In what follows, $X, Y \in \{U, U', W, W'\}$.

Before describing the detailed computation done when the flow for v reaches w, let us look at the subset of T visited by the flow for v (call it $Region(v)$). The flow for v obviously does not visit the proper subtree of v in T, and it obviously does visit every w on the v-to-*root* path in T. For every such w, it may also visit a portion of the subtree of the child

of w (call it u) which is not an ancestor of v; the portion so visited induces a subtree of T rooted at u. If v' is the parent of v then clearly $Region(v') \subseteq Region(v)$ and, if the flow for v' reaches w at (say) stage k, then the flow for v will reach that same w at stage $k+1$.

When the flow for v enters w, $w \neq v$, we obtain the X_v-to-Y_w path lengths. These path lengths are available from the pre-processing stage if w is an ancestor of v, but otherwise they must be computed—we compute them using the Bridging Lemma (Lemma 14). The details of this computation are tricky. When v's flow enters w from w's parent, it can do so under one of two possible *modes* of operation (call them mode 1 and mode 2): mode 1 when $|R_{parent(v)}| \leq |R_w|$, and mode 2 when $|R_w| < |R_{parent(v)}|$. Note that the concept of mode is undefined for a flow that has just entered a node from its child; if v's flow enters w from a child u of w, then v's flow at w has no mode associated with it, and u is an ancestor of v. Observe that, as a result of the definitions of modes 1 and 2, we have the following:

- If the flow for v is at w, then at the next stage the flows of v's children will enter w in mode 1.
- If the flow for v is at w in mode 1, then at the next stage it can go to a child of w in mode 1 or mode 2.
- If the flow for v is at w in mode 2, then at the next stage it can go to a child of w in mode 2 only.
- If the flow for v is at w in mode 2, then $|R_w| = O(|R_v|)$ and, furthermore, that flow will finish visiting w's subtree in $O(1)$ stages.

Obviously, if at stage k, the flow for v is simultaneously at w and w', then its mode at w might be different from its mode at w'.

In order to compute the desired X_v-to-Y_w path lengths, the flow for v gets help from a piece of *preparatory information* that enables it to use the Bridging Lemma; this preparatory information consists of either (i) the $B(Q_v)$-to-Y_w path lengths (if v's flow enters w in mode 1), or (ii) the X_v-to-$B(Q_w)$ path lengths (if v's flow enters w in mode 2). In case (i), this preparatory information is either obtained from $parent(v)$ (if v's flow enters w in mode 1), or is available from the pre-processing (if v's flow enters w from a child of w). In case (ii), the preparatory information comes from v itself (it would have obtained that information at the previous stage). Of course, the assumption that the preparatory information is already available to v as its flow enters w places an extra burden on v: that of computing the

Figure 13: Illustrating the computation of (a) mode 1, and (b) mode 2.

preparatory information that *it will be required to supply at the next stage*; it will supply the information to each of its children u' (because the flow for u' will enter w in mode 1 at the next stage), or it will supply the information to itself (if its own flow will enter a child of w in mode 2 at the next stage). Below we give the details of the computations performed in each of these two modes.

In what follows, suppose the flow for v has just entered w, at stage $k = l(v, w)$. We must prove that we can compute the X_v-to-Y_w path lengths *and* that we can compute the preparatory information to help perform the next stage $k + 1$. The proof is by induction on k, the basis ($k = 1$) being straightforward (since $w = parent(v)$ in that case, and hence all of the needed information is trivially available). The details for the induction step follow. We distinguish two cases, based on the mode in which v's flow has entered w.

Mode 1. $|R_{parent(v)}| \leq |R_w|$: then it must have been the case that, at stage $k - 1$, the flow for $parent(v)$ had already reached w and (by the induction hypothesis) had computed (for its children's future benefit) the $B(Q_v)$-to-Y_w path lengths information. It should be clear that this information (available after stage $k - 1$ at $parent(v)$) enables us to use the Bridging Lemma (Lemma 14) for computing the X_v-to-Y_w path lengths (see Figure 13 (a)), in $O(\log |R_v|)$ time and $O(|R_v||R_w|)$ work.

Now v must compute, for the benefit of each of its own children, say u', the preparatory information that u' will need at the next stage $k + 1$, namely, the $B(Q_{u'})$-to-Y_w path lengths information (note that the flow for u' will enter w in mode 1). But this information is readily available, from the knowledge of the $B(Q_v)$-to-Y_w and the X_v-to-Y_w path lengths information.

Finally, v checks whether its flow will next enter a child u of w in mode 2 and, if so, it

collects the preparatory information that it will then need at the next stage $k+1$, namely, the $B(Q_u)$-to-X_v path lengths. We say "collect" rather than compute, because this information is already available, by the following argument. WLOG, assume $Q_u = Env(LeftR_w)$. The portion of $B(Q_u)$ that is interior to Q_w consists of U'_w and a portion of U_w, and the path lengths between these and X_v have just been computed. We claim that the path lengths between X_v and $B' = B(Q_u) - U'_w - U_w$ had been computed earlier. To see this, first observe that every point $p \in B'$ is either (i) in $B(Q_{lca(u,v)})$ where $lca(u,v)$ is the lowest common ancestor of u and v in T, or (ii) in Y_z for some z on the w-to-$lca(u,v)$ path in T. In case (i) we already know the p-to-X_v path lengths because of the pre-processing. In case (ii), we also know the X_v-to-Y_z path lengths information, because the flow for v has already reached w, and hence had earlier reached z.

Mode 2. $|R_w| < |R_{parent(v)}|$: this implies that w is not an ancestor of $parent(v)$, and that v's flow entered w from $parent(w)$ at the previous stage $k-1$. We claim that v already knows the X_v-to-$B(Q_w)$ path lengths information. To see this, first observe that, if $parent(w)$ is an ancestor of v, then that information is already available from the pre-processing. Otherwise, by the induction hypothesis, v's flow must have prepared that information, when it was at $parent(w)$ at stage $k-1$, for its own use at stage k. The availability of this information implies that we can use the Bridging Lemma (Lemma 14) to compute the X_v-to-Y_w path lengths information, where our v (resp., w) plays the role of the lemma's w (resp., v) (see Figure 13 (b)). Note that as a by-product of this computation, we now know the X_v-to-$B(Q_u)$ path lengths information for each child u of w, and this is precisely the preparatory information that may be needed by v's flow for the next stage, in case v's flow enters u (as already noted, it would do so in mode 2).

We now claim that we can also easily collect, for every child u' of v, the $B(Q_{u'})$-to-Y_w path lengths, which is precisely the preparatory information that is needed by the flow of u' for the next stage, when that flow enters w in mode 1. To prove the claim, assume WLOG that $Q_{u'} = Env(LeftR_v)$. The portion of $B(Q_{u'})$ that is interior to Q_v consists of U'_v and a portion of U_v, and the path lengths between these and Y_w have just been computed. We claim that the path lengths between Y_w and $B' = B(Q_{u'}) - U'_v - U_v$ had been computed earlier. To see this, first observe that every point $p \in B'$ is either (i) in $B(Q_{lca(v,w)})$ where $lca(v,w)$ is the lowest common ancestor of v and w in T, or (ii) in X_z for some z on the $parent(v)$-to-$lca(v,w)$ path in T. In case (i) we already know the p-to-Y_w path lengths because of the pre-processing. In case (ii), we also know the Y_w-to-X_z path

lengths information, because the flow for w has already reached $parent(v)$ and hence had earlier reached z.

To analyze the work complexity of the above scheme, observe that the work done, when w is visited by the flow for v, is $O(|R_v||R_w|)$. Hence the total work is $O(\sum_{v\in T}\sum_{w\in T}|R_v||R_w|)$ $= O(\sum_{v\in T}|R_v|(n\log n)) = O(n^2\log^2 n)$ (where we made use of the fact that $\sum_{w\in T}|R_w| =$ $O(n\log n)$).

Of course, we can collect the lengths of the paths between the points in $V_R \cup B(P)$, which we just computed, into a single $O(n)\times O(n)$ lengths matrix.

6.4 Path Lengths with Arbitrary Query Points

We point out that, given the lengths matrix computed for the case of the V_R-to-V_R paths, we can augment this structure with two planar subdivisions so that we are able to handle a path length query between two *arbitrary* endpoints in $O(\log n)$ time using one processor. We begin with the case of queries with only one arbitrary endpoint, the other endpoint being in V_R, and then we later extend it to the case of two arbitrary endpoints.

Recall that one of the by-products of the previous V_R-to-V_R length matrix computation is the $X(p)$ paths for all $p\in V_R$ and all $X = NE, NW, \ldots$, etc. Given such an $X(p)$ path for a $p\in V_R$, we can use one processor to do a logarithmic time binary search on the path. However, we shall need to do binary search on such paths originating from an *arbitrary* point p (not in V_R). For such a p, the (e.g.) $NE(p)$ path is not explicitly available, but it could easily be obtained *if* we knew which obstacle is first encountered by an upward ray-shooting from p. We can easily perform such a ray-shooting query in logarithmic time and one processor, provided we do the following pre-processing. The horizontal (resp., vertical) trapezoidal edges of V_R, together with the obstacles' boundaries, define an $O(n)$-vertex planar subdivision H_1 (resp., H_2). We pre-process H_1 (resp., H_2) as in [4], in $O(\log n)$ time and $O(n)$ processors, so that it can support a point location query in $O(\log n)$ time with one processor. This enables one processor to determine, in $O(\log n)$ time, which obstacle is first encountered by a horizontal (resp., vertical) ray-shooting from an arbitrary query point p by using H_1 (resp., H_2).

Assume the path length query is between points p and q where, WLOG, $x(q)\le x(p)$ and $y(q)\le y(p)$. If p is arbitrary and $q\in V_R$, then we first check whether p lies above or below $NE(q)$; assume it lies below (the other case is symmetrical). We then perform a leftward ray-shooting query from p. If the ray intersects $NE(q)$ before it hits an obstacle,

then we are done because the path length from p to q is simply $d(p,q)$ (since there is a q-to-p staircase). Otherwise let $e = \overline{q_1 q_2}$ be the (vertical) obstacle edge encountered by the ray-shooting. The length of a shortest q-to-p path is the smaller of the following: (i) $d(p,q_1)$ + the q_1-to-q path length, and (ii) $d(p,q_2)$ + the q_2-to-q path length (recall that the q_1-to-q and q_2-to-q path lengths are readily available, since $q, q_1, q_2 \in V_R$). That the length we seek is the smaller of (i) or (ii) is easy to establish and was in fact proved in [11].

If both p and q are arbitrary, then we first obtain $NE(q)$ in $O(\log n)$ time using one processor, by doing an upward ray-shooting from q, etc. We then proceed exactly as in the previous case, except that we need to use the method of the previous paragraph to compute the lengths of the shortest q_1-to-q and q_2-to-q paths.

7 Path Lengths When $|P| >> |R|$

In this section we consider the case when the polygon P containing the n obstacles has many more vertices than n, that is, $|P| = N >> |R| = n$. So suppose that $|R| = o(|P|)$. We can avoid a term quadratic in N in the work complexity by building a data structure for an *implicit* representation of the path lengths. The method we show here works for any of the versions of the problem we considered earlier, and results in $O(\log N + \log^2 n)$ time and $O(N + n^2 f(n))$ work complexities where $f(n) = 1$ in the $B(P)$-to-$B(P)$ case, and $f(n) = \log n$ in the $B(P)$-to-V_R case. This implicit representation allows us to still use one processor to achieve constant time for a length query whose endpoints are in $B(P) \cup V_R$. The idea is to partition *Bound*(P) into eight *chunks*, each of which is a contiguous portion of *Bound*(P). Each of the eight chunks has associated with it an $O(n)$-vertex unbounded staircase which separates that chunk from the interior of *Env*(R), and that is used to answer queries relevant to that chunk. Since each such staircase has $O(n)$ vertices, we can use the algorithms of the previous sections to process it, that is, to compute length information about paths that have an endpoint on that staircase.

The way we partition *Bound*(P) is by drawing an infinite horizontal (resp., vertical) line from each of the highest and lowest (resp., leftmost and rightmost) edges of *Env*(R). These four lines induce a partition of *Bound*(P) into at most eight connected components, each of which is one of the above-mentioned chunks. We call these the *top, north-east*, $\ldots$, etc. chunks (in clockwise order), respectively (see Figure 14). It is easy to find, for each point in $B(P)$, to which chunk it belongs. We explain how to process the top chunk and

Figure 14: Illustrating the partition of $Bound(P)$.

the north-east one, since the others are obviously analogous. We only consider the shortest paths that are nontrivial in the sense that they link two endpoints that are on segments that do not horizontally or vertically "see each other". The trivial shortest paths are easily handled as explained earlier in Section 4, specifically, in the Discretization Lemma (Lemma 7).

For the top chunk, we let K be the set of vertical projections of the points of $B(Env(R))$ on the horizontal line H defining that chunk. It is obvious that for any vertex p of P in the top chunk, a nontrivial shortest path from p to anywhere below H can be "deformed", without any increase in its length, so that it goes through a point of K, and hence the lengths of paths to the points in K implicitly represent the lengths of all paths to the top chunk.

For the north-east chunk, we project horizontally as well as vertically on that chunk the points of $B(Env(R))$; let K be the set of these $O(n)$ projection points. Let C be $MAX_{NE}(K)$. We must prove that any nontrivial path from a vertex p of P on the north-east chunk which crosses C can be deformed, without any increase in its length, so that it goes through a vertex of C. Let p be any vertex on the north-east chunk, and let q (resp., q') be the point of K that is immediately after (resp., before) p in the linear ordering of that chunk's points. Note that q and q' are not adjacent vertices on C, since there is a vertex q'' of C between them (by definition of the $MAX_{NE}(K)$). Now, consider any nontrivial path to p. Since there is no point of $B(Env(R))$ whose horizontal or vertical projection on the north-east chunk falls in between q and q' in K, it follows that any such path must go below one of $\{q,q'\}$, in which case we can deform it to go through one of $\{q,q'\}$ (say, q) or through q''. Hence the lengths of paths to the vertices of C implicitly represent the lengths of all paths to the

north-east chunk.

To achieve constant query time, we must have associated, in a pre-processing stage, each such p with q and q', something which is easily done by a parallel merging [35] and a parallel prefix [18, 19].

8 Computing the Actual Paths

In this section we present a parallel algorithm for building a data structure that enables us to report an actual shortest path (rather than just its length) between the query points, within $O(\log n)$ time and $O(\log n + k)$ work, where k is the number of segments on that path. Assuming that the structure for querying path lengths is available (computed as in Section 6), the algorithm constructs the data structure for the actual path queries in an additional $O(\log n)$ time and $O(n^2)$ work. We use the same terminology as in Section 6.

The data structure for the path queries consists of: (i) $|V_R|$ shortest path trees, each of them rooted at one of the vertices in V_R, (ii) the two planar subdivisions H_1 and H_2 of Subsection 6.4, and (iii) the $X(v)$ paths for each $v \in V_R$ and $X = NE, NW, \ldots$, etc.

We already discussed the computation of the $X(v)$ paths (in Subsection 6.1), and that of the two planar subdivisions H_1 and H_2 (Subsection 6.4). Hence we need only show how to compute a shortest path tree for every vertex in V_R, and how to use these shortest path trees to process a path query in parallel.

The shortest path trees are computed using the following information: (1) the V_R-to-V_R lengths matrix, containing the lengths of paths between the vertices in V_R (computed in Subsection 6.3), (2) the two planar subdivisions H_1 and H_2, (3) the $X(v)$ paths for each vertex $v \in V_R$, (4) two copies of V_R, one sorted by x coordinates and the other by y coordinates, (5) for every $w \in V_R$, the obstacle (if there exists one) that is hit by a horizontal leftward (resp., rightward) ray-shooting from w, and the obstacle (if there exists one) that is hit by a vertical upward (resp., downward) ray-shooting from w (note that using H_1 and H_2, all these obstacles for a vertex w can be found in $O(\log n)$ time and one processor), and (6) for each edge e of the obstacles, the set of the vertices in V_R whose ray-shootings hit e, denoted as $Hit(e)$, sorted according to where their rays hit e (for example, if e is the right edge of an obstacle, then $Hit(e)$ is the set of vertices in V_R whose horizontal leftward ray-shootings hit e, and $Hit(e)$ is sorted by y coordinates) (note that all the $Hit(e)$ sets can be obtained in $O(\log n)$ time and $O(n \log n)$ work).

We now show how to use the above information in (1)–(6) to construct, in an additional $O(\log n)$ time and linear work, a shortest path tree rooted at a vertex $v \in V_R$. For every $w \in V_R - \{v\}$, we associate a "parent" pointer with w as follows. WLOG, assume that $w \in V_R - \{v\}$ such that $x(v) \leq x(w)$, $y(v) \leq y(w)$, and w is below $NE(v)$; note that in this case, the shortest path between v and w is monotone with respect to the x-axis (see [11] for a proof). If the horizontal leftward ray-shooting from w crosses $NE(v)$ before reaching an obstacle, then a shortest path from w to v is via $NE(v)$; we then let w have an associated pointer to the segment on $NE(v)$ at which the ray from w crosses $NE(v)$. If the ray from w does not cross $NE(v)$, then let u_1 and u_2 be the two vertices of the right edge of the obstacle hit by the ray; using the V_R-to-V_R lengths matrix, we can easily decide whether a shortest path from w to v is via u_1 or via u_2 (say it is via u_1), and we then let w have an associated pointer to u_1. Also we let the segments of each $X(v)$ path be directed toward v.

This computation for vertex v results in a directed graph of $O(n)$ edges and vertices, whose vertices are the union of the vertices in V_R and the vertices of the $X(v)$ paths. This graph is a tree rooted at v because every vertex in the graph except v has exactly one outgoing edge (the pointer to its parent) and no cycle can occur in this directed graph because of the monotonicity property of the shortest paths [11] (recall that this monotonicity states that the only shortest paths we need to consider are those that are monotone with respect to one of the two coordinate axes). Therefore, we have obtained a shortest path tree rooted at v.

It follows that the computation of all the $O(n)$ shortest path trees whose roots are the vertices in V_R can be done in an additional $O(\log n)$ time and $O(n^2)$ work.

Next we discuss how to pre-process the shortest path trees, so that each tree can support a shortest path query between the vertex of V_R stored in the root of the tree and any vertex in V_R. We restrict our attention to the case where both query points are vertices in V_R, because the case of arbitrary query points can be reduced to it in a way similar to the one we used for computing path lengths of arbitrary query points (see Subsection 6.4).

We pre-process each shortest path tree so that the following type of queries can be quickly answered: given a vertex v in the tree and a positive integer i, find the i-th vertex on the path from v to the root of the tree. Such queries are called *level-ancestor queries* by Berkman and Vishkin [5], who gave efficient parallel algorithms for pre-processing rooted trees so that the level-ancestor queries can be answered quickly. The work of Berkman and Vishkin [5, 6] shows (implicitly) that a level-ancestor query can be handled sequentially in

constant time, after a logarithmic time and linear work pre-processing in the CREW-PRAM model. The pre-processing of the shortest path trees is done by simply applying the result of Berkman and Vishkin to each of the $O(n)$ trees, in totally $O(\log n)$ time and $O(n^2)$ work.

For the sake of processor assignment in reporting paths, we also need to compute the number of segments on the actual shortest path which is to be reported. Suppose a shortest path between vertices v and w in V_R is to be reported. The number of segments on such a v-to-w path can be obtained from the depth of w in the shortest path tree rooted at v; it is known that the depths can be computed within the required complexity bounds by using the Euler Tour technique [36].

To report an actual shortest path between vertices v and w in V_R, we do the following. First, we go to the shortest path tree rooted at (say) v, and find the number of segments on the path in the tree from node w to the root v. Let that number be k. The w-to-v path in the tree corresponds to a geometric shortest path between v and w, which we must report. We do so by performing, in parallel, $\lceil k/\log n \rceil - 1$ level-ancestor queries, using node w and integers $\lceil \log n \rceil, 2\lceil \log n \rceil, \ldots, (\lceil k/\log n \rceil - 1)\lceil \log n \rceil$. Each query is handled by one processor in $O(1)$ time. These queries cut the w-to-v path into $\lceil k/\log n \rceil$ pieces of $O(\log n)$ segments each. Finally, we report the $\lceil k/\log n \rceil$ pieces of the path in parallel by assigning one processor to output each piece of the path sequentially.

9 A Note on the Sequential Time Complexity

In this section we make a fairly straightforward observation about the sequential time complexity of the problem we considered (but one that, to the best of our knowledge, has not yet been documented). We sketch an $O(n^2)$ time sequential algorithm for building the data structure that supports the fast processing of the length and path queries (i.e., $O(\log n)$ time for a length query, and $O(\log n + k)$ time for a path query, where k is the number of segments on the path reported). In this sequential algorithm, we take a *topological sort* [2] approach, which is very different from the divide-and-conquer approach used in our parallel algorithms.

We only discuss how to compute the V_R-to-V_R matrix of path lengths, because we have shown (in Sections 6 and 8) that the other components of the data structure can be computed in $O(n^2)$ work (hence $O(n^2)$ sequential time). Recall that these components are the two planar subdivisions H_1 and H_2, the $X(v)$ paths for every $v \in V_R$, and the shortest

path trees rooted at the vertices in V_R, where $X = NE, NW, \ldots$, etc.

Note that there is a sequential algorithm in [11] that optimally solves the single source case of the problem for computing rectilinear shortest paths avoiding rectangular obstacles. The algorithm in [11] uses the plane sweeping technique. This algorithm can be used to compute, in $O(n \log n)$ time, the lengths of the shortest paths between a chosen vertex v in V_R (designated as the fixed source point) and the vertices in $V_R - \{v\}$. Hence the V_R-to-V_R lengths matrix can be obtained by simply applying the algorithm [11] $O(n)$ times (each time a different vertex in V_R is designated as the fixed source point), in totally $O(n^2 \log n)$ time.

The $O(n^2)$ time algorithm is based on the geometric observations given in [11]. The only thing we do differently is that, when computing the path lengths between a fixed vertex v and the vertices in $V_R - \{v\}$, we do not use plane sweeping. Rather, we do topological sorts [2] on $O(n)$ directed acyclic graphs of size $O(n)$ each. These directed graphs will be built using trapezoidal decomposition [32] and the $X(v)$ paths for all $v \in V_R$.

First we show how to build the $O(n)$ directed graphs. For a vertex $v \in V_R$, there are four directed acyclic graphs associated with it. Consider the shortest paths between v and the vertices in $V_R - \{v\}$. The four graphs of v correspond to the following four cases of the shortest paths: (i) those monotone with respect to the x-axis and with v as their *left* endpoints, (ii) those monotone with respect to the x-axis and with v as their *right* endpoints, (iii) those monotone with respect to the y-axis and with v as their *upper* endpoints, and (iv) those monotone with respect to the y-axis and with v as their *lower* endpoints. We only show how to compute for case (i) (the other cases are handled similarly). Let V_R be given sorted by y coordinates.

Suppose that we already know the following information: for the right edge e of each obstacle, the vertex set $Hit(e)$ (recall that this is the set of vertices in V_R whose horizontal leftward ray-shootings hit e). (Computing these sets is done during the pre-processing, by using trapezoidal decomposition [32].) Let u_1 and u_2 be the two vertices of e. For each $w \in Hit(e)$, the path length between w and u_1 (resp., w and u_2) is simply $d(u_1, w)$ (resp., $d(u_2, w)$) and can be trivially computed in $O(1)$ time.

It has been shown in [11] that a shortest path between v and a point p is of case (i) if p is on or is to the right of $NE(v) \cup SE(v)$. We do the following. (1) Find all the vertices in V_R that are on or to the right of $NE(v) \cup SE(v)$; this can be easily done in $O(n)$ time by merging V_R (sorted by y coordinates) and $NE(v) \cup SE(v)$. Let the set of the vertices

that are on or to the right of $NE(v) \cup SE(v)$ be $Right(v)$. $Right(v)$ is the vertex set of the graph. (2) For every vertex $u \in Right(v)$ whose horizontal leftward ray-shooting crosses $NE(v) \cup SE(v)$ before reaching an obstacle, compute the length of its path to v, which is simply $d(v, u)$ (note: there will be no incoming edge for such a vertex u in the graph). (3) For every vertex $w \in Right(v)$ whose horizontal leftward ray-shooting does not cross $NE(v) \cup SE(v)$, let e be the right edge of an obstacle such that $w \in Hit(e)$, and let u_1 and u_2 be the two vertices of e (u_1, $u_2 \in Right(v)$); associate with u_1 (resp., u_2) a pointer to w and assign the pointer a weight equal to $d(u_1, w)$ (resp., $d(u_2, w)$) (note: w has exactly two incoming edges in the graph, one from u_1 and the other from u_2). The construction of this graph for vertex v clearly requires $O(n)$ time.

The directed graph for vertex $v \in V_R$ so constructed is acyclic due to the monotonicity property of the shortest paths in case (i), and it obviously has $O(n)$ vertices and directed edges. The undirected version of the graph may have more than one connected component. A shortest v-to-w path in it, when $w \in Right(v)$, corresponds to a shortest geometric path between v and w. The single-source shortest paths problem in such a graph can easily be solved in linear time, since it is acyclic. Therefore the V_R-to-V_R path lengths matrix can be computed in $O(n^2)$ time.

10 Conclusion

We have obtained efficient parallel algorithms for building a data structure that supports fast processing of queries about the lengths of the shortest paths between arbitrary points, and about the actual paths.

The techniques involved in the solution include: (i) efficiently finding a "staircase separator" and using it to guide the recursion, (ii) reducing the transitive closure computation in the "conquer" stage to a *constant* number of $(\min, +)$ matrix multiplications (instead of the usual logarithmic number of matrix multiplications), and (iii) showing that the matrices being multiplied in the "conquer" stage have a special structure that enables us to avoid the super-quadratic work bottleneck that is usually the price paid for doing parallel matrix multiplication. In addition to the above techniques (which are likely to be useful in other contexts), we used a number of observations that are specific to this particular kind of path problems. We achieved (ii) and (iii) by partitioning the obstacles' boundaries in a way which ensures that the resulting path length matrices we use have a monotonicity property that

is apparently absent before applying our partitioning scheme. The most general version of our algorithm required a novel pipelining of the computation up and down the recursion tree, with $O(n)$ computational "flows" that originate from all nodes and proceed only to the nodes whose associated problem size is *larger* than that of the flow's origin.

Acknowledgement. The authors are grateful to the referees for their helpful comments. A preliminary version of this work appeared in the *Proceedings of the 2-th Annual ACM Symposium on Parallel Algorithms and Architectures* (1990).

References

[1] A. Aggarwal and J. Park. "Notes on searching in multidimensional monotone arrays (preliminary version)." *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science,* 1988, pp. 497–512.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

[3] A. Apostolico, M. J. Atallah, L. L. Larmore, and H. S. McFaddin. "Efficient parallel algorithms for string editing and related problems." *SIAM J. Comput.* 19 (5) (1990), pp. 968–988.

[4] M. Atallah, R. Cole, and M. Goodrich. "Cascading divide-and-conquer: a technique for designing parallel algorithms." *SIAM J. Comput.* 18 (3) (1989), 499–532.

[5] O. Berkman and U. Vishkin. "Finding level-ancestors in trees." Tech. Rept. UMIACS-TR-91-9, University of Maryland, 1991.

[6] O. Berkman and U. Vishkin. Personal communication.

[7] R. P. Brent. "The parallel evaluation of general arithmetic expressions." *J. of the ACM* 21 (2) (1974), 201–206.

[8] J. Chen and Y. Han. "Shortest paths on a polyhedron." *Proc. 6rd Annual ACM Symp. Computational Geometry,* 1990, pp. 360–369.

[9] K. L. Clarkson, S. Kapoor, and P. M. Vaidya. "Rectilinear shortest paths through polygonal obstacles in $O(n(\log n)^2)$ time." *Proc. 3rd Annual ACM Symp. Computational Geometry,* 1987, pp. 251–257.

[10] R. Cole. "Parallel merge sort." *SIAM J. Comput.* 17 (4) (1988), 770–785.

[11] P. J. de Rezende, D. T. Lee, and Y. F. Wu. "Rectilinear shortest paths in the presence of rectangles barriers." *Discrete Comput. Geom.* 4 (1989), 41–53.

[12] H. ElGindy and M. Goodrich. "Parallel algorithms for shortest path problems in polygons." *The Visual Computer* 3 (6) (1988), 371–378.

[13] H. ElGindy and P. Mitra. "An efficient parallel for computing the orthogonal shortest path among rectangular obstacles." Manuscript, January 1991.

[14] M. T. Goodrich, S. B. Shauck, and S. Guha. "Parallel methods for visibility and shortest path problems in simple polygons (Preliminary version)." *Proc. 6th Annual ACM Symp. Computational Geometry,* 1990, pp. 73–82.

[15] S. Guha and Q. Stout. Personal communication, July 1990.

[16] L. J. Guibas and J. Hershberger. "Optimal shortest path queries in a simple polygon." *Proc. 3rd Annual ACM Symp. Computational Geometry*, 1987, pp. 50–63.

[17] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. "Linear time algorithms for visibility and shortest paths problems inside triangulated simple polygons." *Algorithmica*, 2 (1987), 209–233.

[18] C. P. Kruskal, L. Rudolph, and M. Snir. "The power of parallel prefix." *IEEE Trans. on Computers* C-34 (1985), 965–968.

[19] R. E. Ladner and M. J. Fischer. "Parallel prefix computation." *J. of the ACM* 27 (4) (1980), 831–838.

[20] R. C. Larson and V. O. Li. "Finding minimum rectilinear distance paths in the presence of barriers." *Networks* 11 (1981), 285–304.

[21] D. T. Lee. "Proximity and reachability in the plane." Ph.D. Thesis, Technical Report ACT-12, Coordinated Science Laboratory, Univ. of Illinois, Nov. 1978.

[22] D. T. Lee, T. H. Chen, and C. D. Yang. "Shortest rectilinear paths among weighted obstacles." *Proc. 6rd Annual ACM Symp. Computational Geometry*, 1990, pp. 301–310.

[23] D. T. Lee and F. P. Preparata. "Euclidean shortest paths in the presence of rectilinear barriers." *Networks* 14 (1984), 393–410.

[24] T. Lozano-Perez and M. A. Wesley. "An algorithm for planning collision-free paths among polyhedral obstacles." *Communications of the ACM* 22 (1979), 560–570.

[25] J. S. B. Mitchell. "Shortest rectilinear paths among obstacles." *Computational and Geometric Aspects of Robotics. Proc. 1st Int. Conf. on Industrial and Applied Math. (ICIAM'87)*, Paris, 1987, pp. 39–84.

[26] J. S. B. Mitchell. "An optimal algorithm for shortest rectilinear path among obstacles." *First Canadian Conference on Computational Geometry*, 1989.

[27] J. S. B. Mitchell. "Planning shortest paths." Ph.D. thesis, Dept. of Operation Research, Stanford Univ., 1986.

[28] J. S. B. Mitchell, D. M. Mount, and Papadimitriou. "The discrete geodesic problem." *SIAM J. Comput.* 16 (1987), pp. 647–668.

[29] J. S. B. Mitchell and Papadimitriou. "Planning shortest paths." *SIAM Conference*, 1985, pp. 15–19.

[30] T. M. Nicholl, D. T. Lee, Y. Z. Liao, and C. K. Wong. "On the X-Y convex hull of a set of X-Y polygons." *BIT* 23 (4) (1983), 456–471.

[31] j O'Rourke, S. Suri, and H. Booth. "Shortest paths on polyhedral surfaces." Manuscript, The Johns Hopkins Univ., 1984.

[32] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985, pp. 151.

[33] J. Reif and J. A. Storer. "3-Dimensional shortest paths in the presence of polyhedral obstacles." *Proc. Foundations of Comput. Sci.*, 1988, pp. 85–92.

[34] M. Sharir and A. Schorr. "On shortest paths in polyhedral spaces." *SIAM J. Comput.* 15 (1) (1986), 193–215.

[35] Y. Shiloach and U. Vishkin. "Finding the maximum, merging, and sorting in a parallel computation model." *J. Algorithms* 2 (1981), 88–102.

[36] R. E. Tarjan and U. Vishkin. "An efficient parallel biconnectivity algorithm." *SIAM J. Comput.* 14 (4) (1985), 862–874.

[37] P. Widmayer, Y. F. Wu, and C. K. Wong. "On some distance problems in fixed orientations." *SIAM J. Comput.* 16 (4) (1987), 728–746.

[38] Y. F. Wu, P. Widmayer, M. D. F. Schlag, and C. K. Wong. "Rectilinear shortest paths and minimum spanning trees in the presence of rectilinear obstacles." *IEEE Trans. on Computers* C-36 (1987), 321–331.

[39] C. D. Yang, T. H. Chen, and D. T. Lee. "Shortest rectilinear paths among weighted rectangles." *Journal of Information Processing,* to appear.

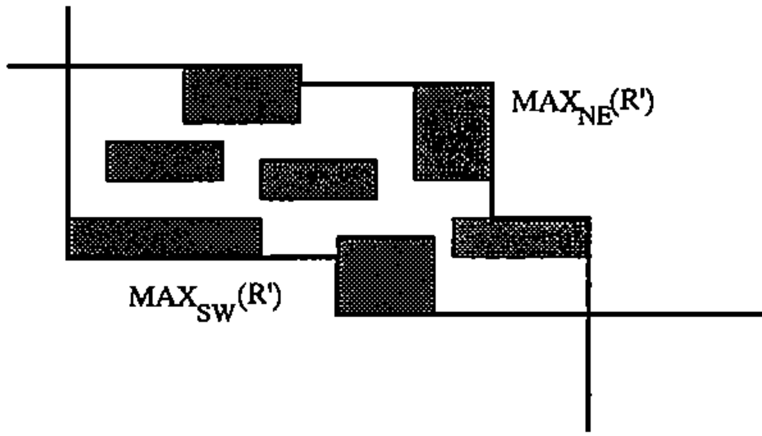


Figure 1

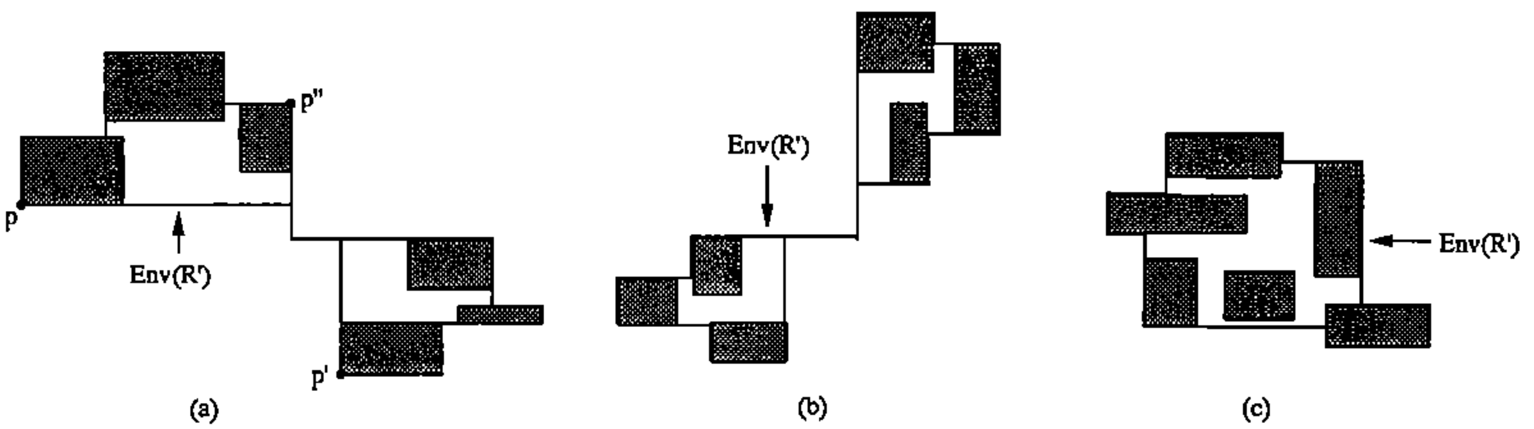


Figure 2

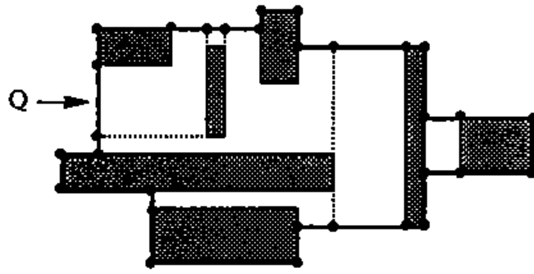


Figure 3

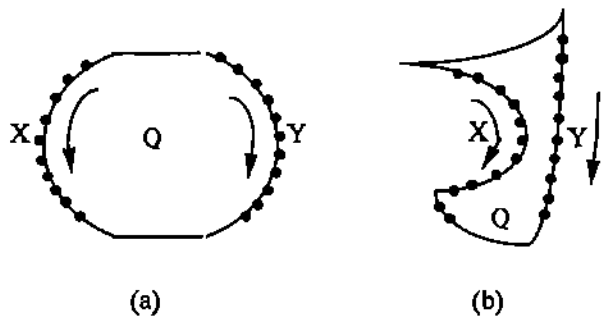


Figure 4

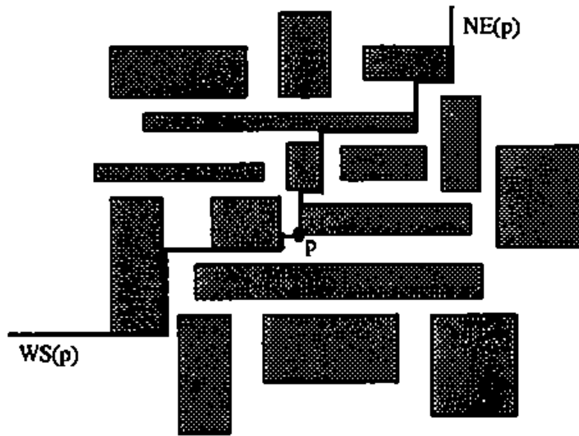


Figure 5

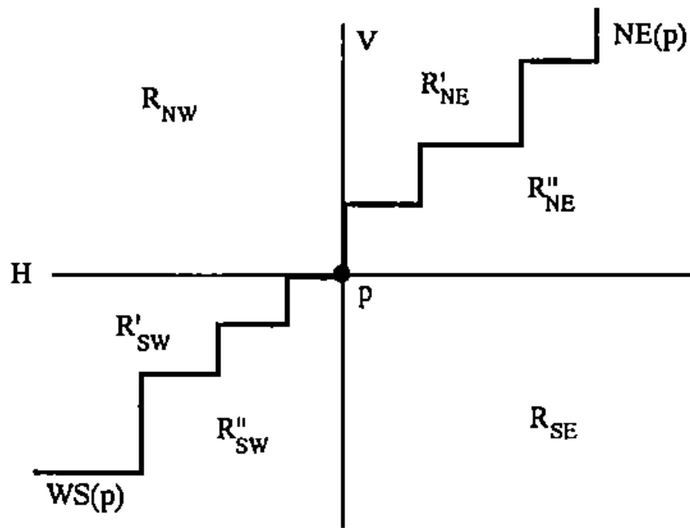


Figure 6

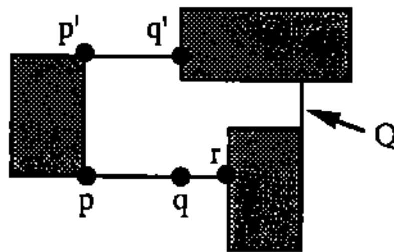


Figure 7

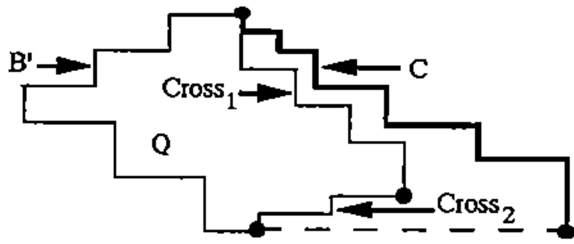


Figure 8

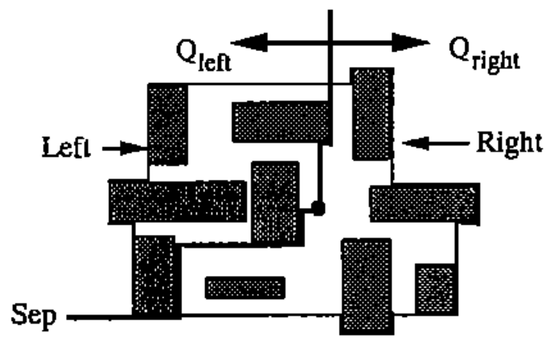


Figure 9

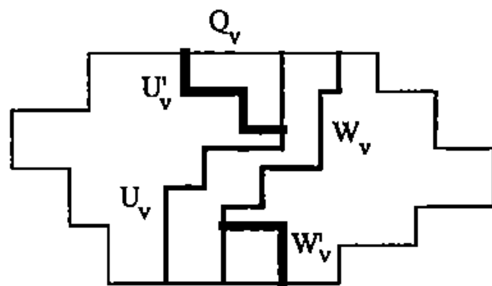


Figure 10

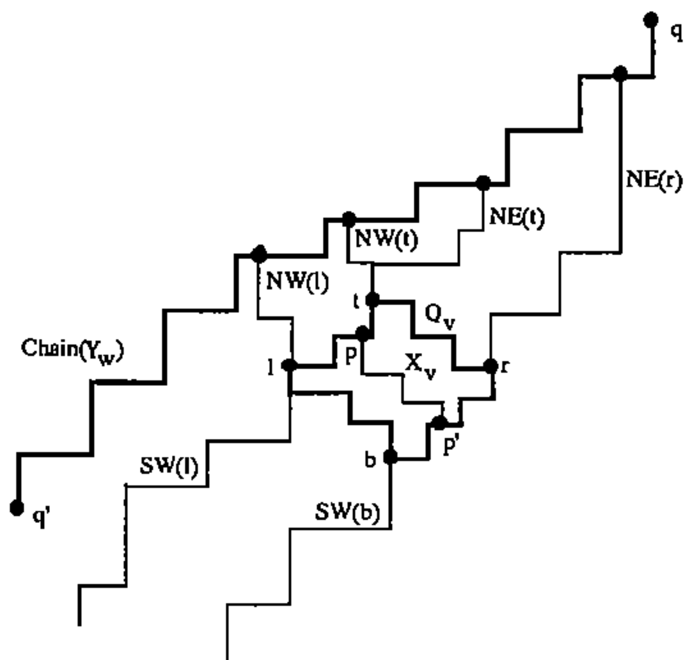


Figure 11

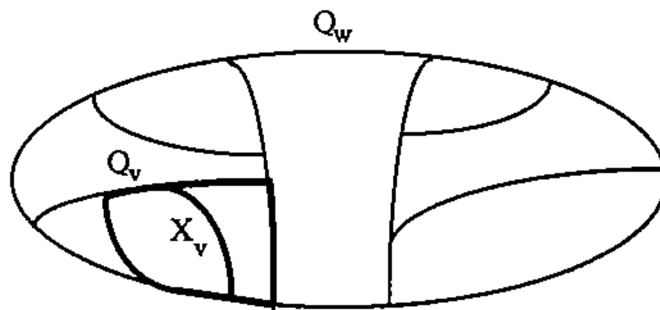
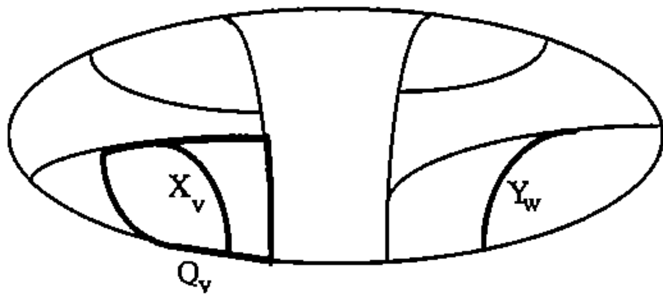
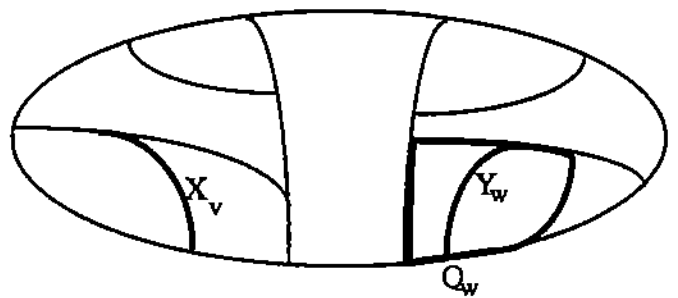


Figure 12



(a)



(b)

Figure 13

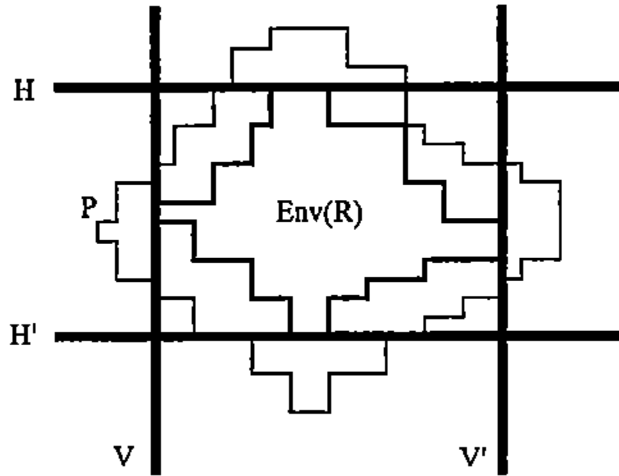


Figure 14