Matrix Searching with the Shortest Path Metric

John Hershberger
DEC/SRC
130 Lytton Avenue,
Palo Alto, California 94301

Subhash Suri
Bellcore
445 South Street,
Morristown, New Jersey 07960

August 16, 1993

Abstract

We present an $O(n)$ time algorithm for computing row-wise maxima or minima of an implicit, totally-monotone $n \times n$ matrix whose entries represent shortest-path distances between pairs of vertices in a simple polygon. We apply this result to derive improved algorithms for several well-known problems in computational geometry. Most prominently, we obtain linear-time algorithms for computing the geodesic diameter, all farthest neighbors, and external farthest neighbors of a simple polygon, improving the previous best result by a factor of $O(\log n)$ in each case.

Key Words: Shortest paths, matrix searching, geodesic diameter, farthest neighbors, geometric matching.

1 Introduction

Matrix-searching is the popular term for a technique introduced by Aggarwal et al. [2] for computing row-wise maxima in a totally monotone matrix. A matrix M is called *totally monotone* if

$$M(i,k) < M(i,l) \implies M(j,k) < M(j,l),$$

for any $i < j$ and $k < l$. Aggarwal et al. [2] discovered the importance of totally monotone matrices and showed that many problems in computational geometry can be formulated as finding maxima in a totally monotone matrix. The problem of finding farthest neighbors in a convex polygon is a prototypical application of this technique; we repeat it here for the benefit of the uninitiated.

Consider a convex polygon P and partition its boundary into two disjoint chains $U = \{u_1, u_2, \ldots, u_p\}$ and $V = \{v_1, v_2, \ldots, v_m\}$. The vertices in each chain are ordered counterclockwise, as shown in Figure 1. Define a $p \times m$ matrix M that encodes pairwise distances between the vertices of U and V: $M(i,j) = d(u_i, v_j)$, for $1 \leq i \leq p$ and $1 \leq j \leq m$. One may easily check that M is totally monotone; the proof follows from the triangle inequality. We observe that a maximum entry in row i of M corresponds to the maximum distance between u_i and any vertex in V. Thus, we can find for all vertices of U a corresponding farthest neighbor in V by computing the row-wise maxima in M.

The main result of Aggarwal et al. [2] says that the row-wise maxima of a totally monotone $n \times n$ matrix can be found with only $O(n)$ comparisons and evaluations of the matrix entries. The matrix is defined implicitly—an entry is evaluated only when needed by the algorithm. If evaluating a matrix entry takes $f(n)$ time, then the row-wise maxima can be computed in time $O(nf(n))$. This result leads to linear-time algorithms for several computational geometry problems set in convex polygons, where the matrix entries correspond to Euclidean distances.

In this paper, we propose a powerful extension of the basic matrix-searching algorithm. We give a linear time algorithm for computing row-wise maxima in a totally monotone matrix whose entries represent *shortest path* distances between pairs of vertices in a simple polygon. The shortest path distance, also called the *geodesic distance*, is the natural distance measure in a simple nonconvex polygon. Observe, however, that unlike the Euclidean distance, the geodesic distance in general cannot be computed in constant time. Unless one precomputes all the distances, which requires too much preprocessing, the best data structure for computing shortest path distances is one due to Guibas and Hershberger [6, 10], which achieves $f(n) = O(\log n)$. (After an $O(n)$ time and space preprocessing step, this data structure computes the geodesic distance between two arbitrary points of a simple polygon in $O(\log n)$ time; n denotes the number of vertices in the polygon.) A straightforward combination of the shortest path query data structure and the basic matrix-searching gives an $O(n \log n)$ time algorithm for computing row-wise maxima of a totally monotone matrix with shortest path distances. Our result in this paper improves the time complexity to optimal $O(n)$.

Our result has applications in computational geometry: we achieve linear time algorithms for several outstanding open problems. The most straightforward application concerns geodesic diameter and all-farthest neighbors. We obtain $O(n)$ time algorithms for them, improving the previous $O(n \log n)$ bound. A similar improvement also follows for the problem of computing the external farthest neighbors in a simple polygon.

A somewhat less obvious application concerns finding a closest visible vertex pair between two simple polygons. Although this problem does not involve the shortest path metric in its formulation, we apply matrix-searching with shortest paths to get a conceptually simple algorithm for the problem that runs in optimal linear time. Previous linear-time algorithms were quite complicated [3, 4]. In a separate paper [12] we apply this result, along with some additional techniques, to obtain a linear-time algorithm for computing a shortest diagonal of a simple polygon, improving the previous $O(n \log n)$ algorithm.

We can also compute weighted-nearest neighbors between two disjoint chains of a simple polygon in linear time; the weighted distance is defined as $d(u_i, v_j) - w(u_i) - w(v_j)$, where $d(u_i, v_j)$ is the shortest path distance, and w is a real-valued weight function. This improves the complexity of computing an optimal matching on the vertices of a polygon from $O(n \log^2 n)$ to $O(n \log n)$ [13], matching the best bound for a convex polygon.

Our paper is organized in seven main sections. In Section 2, we formulate the key problem of our paper, and provide a brief summary of the matrix searching technique. In Section 3, we introduce the basic geometric infrastructure required by our algorithm. The algorithm itself is described in Section 4, and its analysis is presented in Section 5. The applications of our main result are presented in Section 6, and we conclude with some remarks in Section 7.

2 Matrix Searching and the Shortest Path Metric

The centerpiece of our paper is an algorithm for computing farthest neighbors between two chains of a simple polygon under the shortest path metric. In this section, we introduce the problem, define terminology, and provide a brief review of the matrix-searching framework of Aggarwal et al. [2]. Throughout, we assume that P is a simple polygon with n vertices; $\pi(x, y)$ is the shortest path between the points x and y in P; $d(x, y)$ is the length of the path $\pi(x, y)$; and y is a (geodesic) *farthest neighbor* of x if $d(x, y) = \max\{d(x, x') \mid x' \in P\}$.

2.1 Farthest neighbors and the totally monotone matrix

Let $U = (u_1, u_2, \ldots, u_p)$ and $V = (v_1, v_2, \ldots, v_m)$ be two polygonal chains that together partition the vertices of P; the vertices are ordered counterclockwise in each chain, and the total number of vertices is $n = p + m$. We want to compute a farthest neighbor for each vertex of U in the chain V. This problem can be reformulated as a maxima-finding problem in the associated distance matrix.

Let M denote the $p \times m$ matrix that encodes the pairwise distances between the vertices of U and V:

$$M(i, j) = d(u_i, v_j). \quad (1)$$

See Figure 1. Our problem is to compute a maximum entry in each row of M. (The equivalence is clear: $M(i, j)$ is a maximum entry in row i if and only if v_j is a geodesic farthest neighbor of u_i.) We break ties by choosing the leftmost maximum in each row.

We use the matrix-searching framework of Aggarwal et al. [2], which depends crucially on a matrix property called *total monotonicity.* The matrix M is *totally monotone* if

$$M(i,k) < M(i,l) \implies M(j,k) < M(j,l), \tag{2}$$

for any $1 \le i < j \le p$ and $1 \le k < l \le m$. The total monotonicity of M follows from the triangle inequality of the geodesic distance function. Total monotonicity also implies that the leftmost maxima form a "staircase" in the matrix: as we move down the rows, the maximum can only move right, but never left.

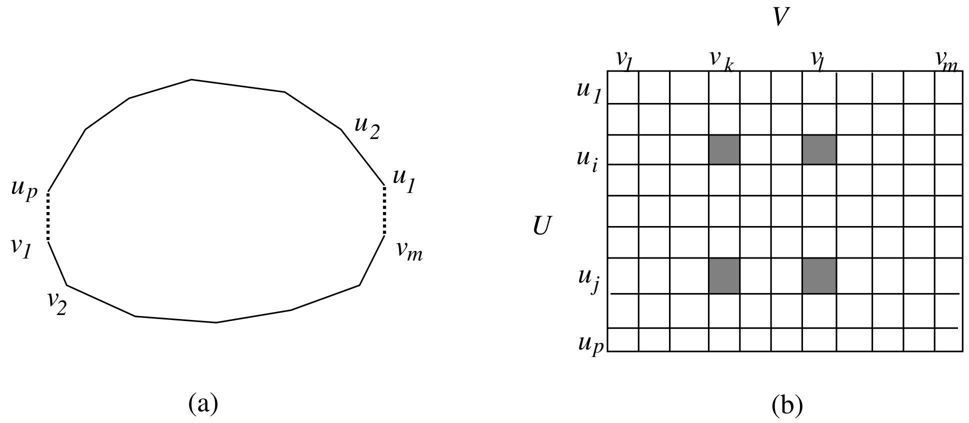

Figure 1: A polygon, two chains, and the corresponding matrix. The triangle inequality implies that $M(i,k)+M(j,l) \ge M(i,l)+M(j,k)$, which in turn shows that M is totally monotone.

The matrix M is defined implicitly: an entry is computed only when needed. The matrix-searching algorithm performs two kinds of operations: evaluation of a matrix entry $M(i,j)$ and comparison between two matrix entries. We use the term *matrix operation* to describe either of these two steps. In general, the comparison step can always be performed in constant time, but the evaluation step can be more complicated. The main result of Aggarwal et al. [2] is the following lemma.

Lemma 2.1 ([2]) *The leftmost maximum in each row of a totally monotone $n \times n$ matrix can be determined using $O(n)$ matrix operations. If each matrix operation takes $f(n)$ time, then the row-maxima problem can be solved in time $O(nf(n))$.*

If we use shortest path queries [6, 10] to compute the entries of M, then $f(n) = O(\log n)$, and we get an $O(n \log n)$ time algorithm. Our contribution in this paper is an algorithm that reduces $f(n)$ to amortized $O(1)$, giving an optimal linear-time algorithm.

2.2 The matrix-searching algorithm

Our algorithm assumes some familiarity with the matrix-searching algorithm of [2]. This section gives a brief overview of the matrix-searching technique; however, the reader is encouraged to refer to the original paper for details.

The matrix-searching algorithm of [2] finds the leftmost maximum in each row of a totally monotone matrix using only a linear number of matrix operations. The algorithm is recursive; it proceeds in $O(\log n)$ phases. Each phase cuts down the dimensions of the matrix by half, after evaluating a linear number of matrix entries. The following pseudocode gives a top-level view of the algorithm.

MSEARCH(M)

(∗ *Compute the position and value of the leftmost maximum in each row of* M ∗)

1. $A \leftarrow$ *REDUCE*(M)
2. **if** $|A| = 1$ **then return** the value of $A[1,1]$ and its position in M
3. $B \leftarrow$ the matrix consisting of *even* rows of A
4. *MSEARCH*(B)
5. *MFILL*(A, B) (∗ Finds maxima in the odd rows of A ∗)
6. Compute the positions and values of the maxima in M from their positions in A.

REDUCE takes a $p \times m$ matrix M, where $m \geq p$, and returns a $p \times p$ submatrix A. (If $m < p$, *REDUCE* just returns M.) The leftmost maxima in the rows of A and M are identical. *REDUCE* achieves this size-reduction using only $O(m)$ matrix operations. In the pseudocode below, A^k refers to the kth column of A.

```
REDUCE (M)
    A ← M;   k ← 1;
    while A has more than p columns do
        case
        A(k,k) ≥ A(k,k+1) and k < p:   k ← k+1.
        A(k,k) ≥ A(k,k+1) and k = p:   Delete column A^{k+1}.
        A(k,k) < A(k,k+1):             Delete column A^k;
                                       if k > 1 then k ← k−1.
        endcase
return A.
```

Figure 2 illustrates the pattern of comparisons performed by *REDUCE*. No entry in the shaded portion of the matrix can be a candidate for the leftmost maxima.

The code above, which is essentially in the form given by Aggarwal et al. [2], operates on a dynamic matrix: when a column is deleted, all columns to its right are renumbered. This property makes it somewhat difficult to identify which entries of the original matrix are examined. Because our algorithm depends crucially on the order in which matrix entries are computed, we rewrite the code for *REDUCE* to clarify the matrix access pattern. The revised version below uses a *predecessor* array (essentially a linked list) to delete columns. Initially *pred*$[i]$ is set to $i-1$ for all i. To delete a column j such that *pred*$[k] = j$, the

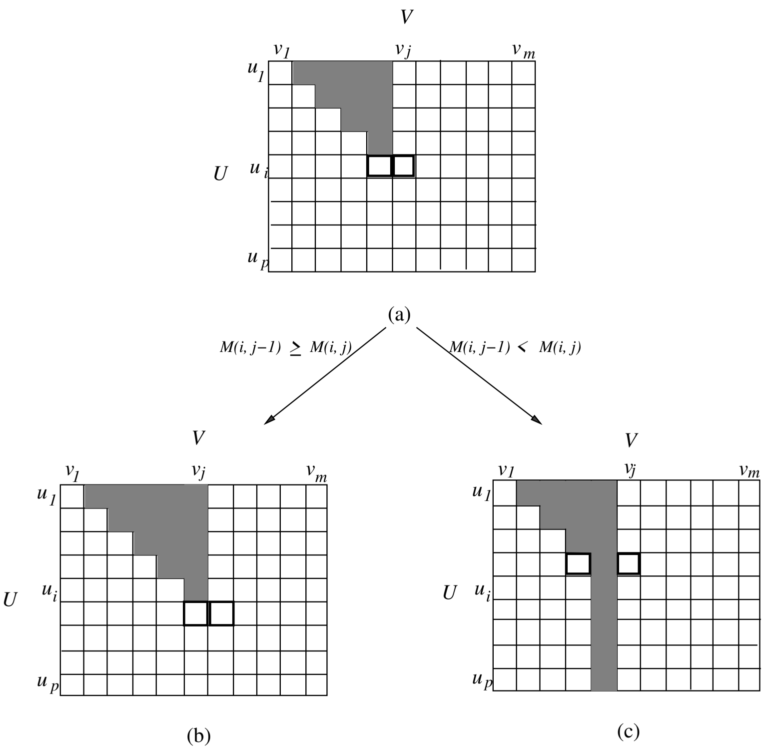

Figure 2: Illustrating *REDUCE*. The current comparison is between $M(i, j-1)$ and $M(i, j)$, shown by the highlighted cells in (a). If $M(i, j-1) \geq M(i, j)$, then we arrive at Figure 2(b): the first i rows of the jth column are eliminated, and the next comparison is between $M(i+1, j)$ and $M(i+1, j+1)$. Otherwise $M(i, j-1) < M(i, j)$, and we arrive at Figure 2(c): column $j-1$ is eliminated entirely. In this case, the search backs up and next compares $M(i-1, j-2)$ and $M(i-1, j)$.

algorithm resets $pred[k] \leftarrow pred[j]$, thereby splicing j out of the list. The following code avoids some of the matrix probes in the version of *REDUCE* above by reusing previously computed values.

NEW-REDUCE (M)
  for $j \leftarrow 1$ **to** m **do** $pred[j] \leftarrow j-1$;
  $ncols \leftarrow m$; $j \leftarrow 2$; $k \leftarrow 1$;
  $value[1] \leftarrow M(1,1)$;
  while $ncols > p$ **do**
    ($*$ Note that $value[pred[j]] = M(k, pred[j])$ $*$)
    if $value[pred[j]] \geq M(k,j)$ **then**
      if $k < p$ **then**
        $k \leftarrow k+1$; $value[j] \leftarrow M(k,j)$;
      else
        $pred[j+1] \leftarrow pred[j]$; $ncols \leftarrow ncols - 1$;
      endif
      $j \leftarrow j+1$;
    else
      $pred[j] \leftarrow pred[pred[j]]$; $ncols \leftarrow ncols - 1$;
      if $k \neq 1$ **then** $k \leftarrow k-1$ **else** $j \leftarrow j+1$ **endif**;
    endif
  endwhile
return the $pred[*]$ linked list.

Lemma 2.2 *The two algorithms REDUCE and NEW-REDUCE are equivalent.*

Proof: We need to prove the correspondence between the chain of $pred[*]$ pointers in *NEW-REDUCE* and the dynamic matrix in *REDUCE*. To this end, define the function $length(j)$ to be the length of the chain of $pred[*]$ pointers starting at j. That is, $length(j) = h$ iff $pred^h[j] = 0$, but $pred^{h-1}[j] \neq 0$.

We maintain the invariant that at the top of each loop, if column k in *REDUCE* is actually column j of the original matrix M, then $length(j) = k$ in *NEW-REDUCE*. This is established by the **for** loop of *NEW-REDUCE*. The main loop of *REDUCE* corresponds to that of *NEW-REDUCE*. At the top of each **while** loop, column j of *NEW-REDUCE* corresponds to column $k+1$ in *REDUCE*, and column $pred[j]$ corresponds to column k.

We must prove that the matrix comparisons in *REDUCE* and *NEW-REDUCE* are equivalent. The main loop of *NEW-REDUCE* explicitly computes the value needed on the right hand side of the comparison in that loop. The value needed on the left hand side of the comparison is referred to as $A(k,k)$ in *REDUCE*; that is, it is the matrix element $M(k,l)$ such that $length(l) = k$ in *NEW-REDUCE* and $l = pred[j]$. So, we must show that $value[l] = M(k,l)$. If *NEW-REDUCE* does not delete column j, it sets $value[j] \leftarrow M(length(j), j)$ just before it advances to column $j+1$. Because *NEW-REDUCE* does not change *length*(l) for any $l < j$, it follows that *value*$[l] = M(k,l) = A(k,k)$. We leave it as an exercise to the reader to prove the exact correspondence between the statements of *REDUCE* and *NEW-REDUCE*. $\square$

In a high-level view of the algorithm, the two versions *REDUCE* and *NEW-REDUCE* are completely equivalent; the latter only fleshes out more details. The *pred*$[*]$ list that *NEW-REDUCE* returns encodes the reduced matrix A returned by *REDUCE*. In an implementation, *MSEARCH* and *MFILL* would also use *pred* pointers; however, in order to preserve a close parallel with the basic algorithm of [2], we will use *NEW-REDUCE* only in the next subsection, and use *REDUCE* in the remainder of the paper.

Finally, the procedure *MFILL*(A, B) computes the leftmost maxima for the odd rows of A, given the locations of the maxima for its even rows; recall that the matrix B consists of precisely the even rows of A. This phase also requires $O(m)$ matrix operations. Let C_{2j} denote the column index of the leftmost maximum in the even rows of A, for $j = 1, 2, \ldots, \lceil p/2 \rceil$. Then total monotonicity implies that C_{2j+1} lies between C_{2j} and C_{2j+2}. The following pseudocode implements *MFILL*. For convenience we set $C_0 = 1$ and $C_{2\lfloor p/2 \rfloor + 2} = m$.

```
MFILL (A, B)
    for i ← 1 to ⌈p/2⌉ do
        max ← A(2i − 1, C_{2i−2});
        C_{2i−1} ← C_{2i−2};
        for j ← C_{2i−2} + 1 to C_{2i} do
            if A(2i − 1, j) > max then
                max ← A(2i − 1, j);
                C_{2i−1} ← j;
            endif
        Evaluate A(2i, C_{2i});   (* Extra work, for accounting purposes *)
    endfor;
```

Our algorithm has exactly the same form as that of Aggarwal et al. [2]; only the details of evaluating a matrix entry are different. In [2], each matrix evaluation takes constant time. In our case, the evaluations are trickier, and require a careful implementation of a shortest path algorithm. To this end, we need to understand better the path followed by the matrix evaluations in the subroutine *REDUCE*.

2.3 Summary of matrix searching

Each phase of the recursive algorithm *MSEARCH* halves the matrix dimensions. The core of the algorithm is *REDUCE*, which eliminates columns of the matrix. In the kth phase of the algorithm, *REDUCE* operates on a submatrix A_k of size at most $n_k \times 2n_k$, where $n_k = \lfloor p2^{-k} \rfloor$. From here on we will assume that A_k is exactly $n_k \times 2n_k$. Matrix-searching theory tells us that *REDUCE* performs at most $3n_k$ matrix operations, and returns an $n_k \times n_k$ submatrix.

We call the vertices corresponding to rows and columns of A_k *active* vertices; the remaining vertices of U and V are called *dead.* Thus, there are n_k active vertices in the U-chain, and $2n_k$ active vertices in the V-chain. Because *MSEARCH* deletes even rows, the active vertices of U are uniformly spaced in U; more precisely, two adjacent active vertices are separated by $2^k - 1$ dead vertices. However, no such regularity necessarily exists among the active vertices of V.

Our algorithm makes critical use of the order in which the matrix-searching algorithm evaluates entries. This order is easier to analyze in *NEW-REDUCE* than in the equivalent routine *REDUCE*. The first matrix entry evaluated by *REDUCE* (or *NEW-REDUCE*) in the kth phase is $A_k[1,1]$. In each new evaluation, either the row or the column index is changed by one. In particular, the movement in the matrix consists of steps that belong to $\{up,\ down,\ right\}$, where *up* and *down* decrement and increment the row-index and *right* increments the column-index. The column-index is never decremented, so there is no *left* movement in the matrix. We can break the sequence of moves into blocks that start with a *right* move. Each *right* move is followed by zero or more *up* moves and at most one *down* move. Thus the sequence of moves satisfies the following regular expression:

$$(\ right\ \ up^*\ \ (down \mid \epsilon)\ )^* \tag{3}$$

where ϵ is the null move. The sequence of evaluations in *MFILL* also satisfies the same regular expression.

3 Geometric Structures

In this section, we introduce the geometric infrastructure necessary for computing the shortest path distances. We describe both the geometric properties of these structures and their implementations. We start with some notation. The shortest path between two points x and y in P is denoted $\pi(x,y)$. The two extreme vertices of the chain V play a distinguished role in our algorithm. We refer to them with the shorthand notation $a = v_1$ and $b = v_m$.

3.1 The shortest path tree

Let T_a and T_b denote the shortest path trees of the polygon P rooted at the vertices $a = v_1$ and $b = v_m$, respectively. (A shortest path tree T_p is the union of shortest paths from p to all the vertices of P.) We precompute an array that stores, for every vertex x of P, the parents of x in T_a and T_b, along with the distances from x to a and b. We first triangulate the polygon in linear time using Chazelle's algorithm [5], and then build the shortest path trees T_a and T_b also in linear time using an algorithm of Guibas et al. [7] or Hershberger and Snoeyink [11].

Lemma 3.1 ([7, 11]) *After linear-time preprocessing, the distances $d(x,a)$ and $d(x,b)$, for any vertex $x \in P$, can be computed in $O(1)$ time.*

Given two arbitrary vertices $x, y \in P$, we let $\alpha(x,y)$ denote the *lowest common ancestor* of x and y in T_a. Similarly, $\beta(x,y)$ denotes the lowest common ancestor of x and y in the tree T_b. In our discussion, we so frequently use lowest common ancestors where one of the vertices is a or b that we also introduce the following abbreviated notation:

$$\begin{aligned}\alpha(x) &= \alpha(x, b) \\ \beta(x) &= \beta(x, a)\end{aligned}$$

In Fig. 3, $\alpha(u_{19}, v_{10}) = v_4$, $\beta(u_{19}, v_{10}) = u_{12}$, $\alpha(v_{10}) = v_4$, and $\beta(v_{10}) = v_{17}$.

We use the linear-time algorithm of Harel and Tarjan [9] or that of Schieber and Vishkin [15] to preprocess our trees for lowest common ancestor queries.

Lemma 3.2 ([9, 15]) *After linear-time preprocessing of T_a and T_b, we can compute $\alpha(x, y)$ and $\beta(x, y)$ in constant time for any two vertices $x, y \in P$.*

3.2 The funnel and the funnel-difference

Consider a vertex $v \in V$ and the two shortest paths $\pi(v, a)$ and $\pi(v, b)$. These two paths may coincide for some distance before they diverge at a vertex z, never to meet again. The *funnel* of v, denoted $F(v)$, is the set of edges in the shortest paths from z, called the *apex* of the funnel, to the lowest common ancestor vertices $\alpha(v)$ and $\beta(v)$:

$$F(v) = \pi(z, \alpha(v)) \cup \pi(z, \beta(v)). \tag{4}$$

Both $\pi(z, \alpha(v))$ and $\pi(z, \beta(v))$ are concave. See Figure 3 for an example of funnels.

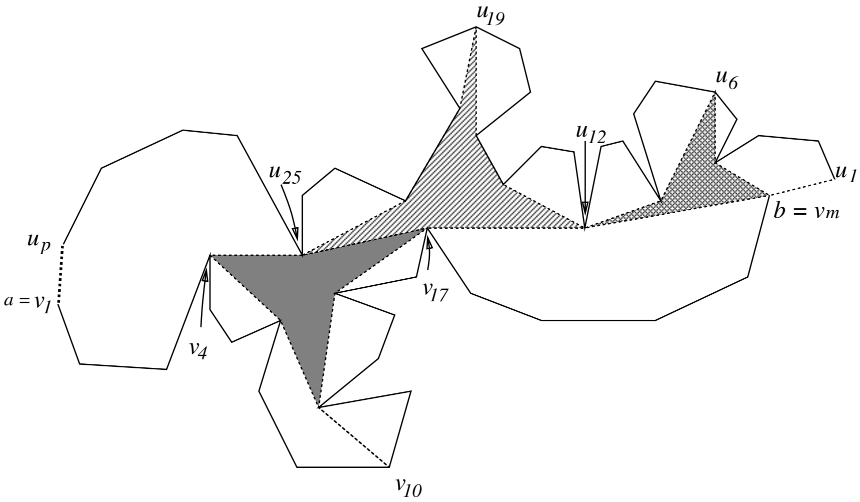

Figure 3: The example shows funnels for three vertices: v_{10}, u_6, and u_{19}. Lowest common ancestors bounding $F(v_{10})$ are v_4 and v_{17}. The apex of $F(v_{10})$ is v_{12}. Similarly, $F(u_6)$ is bounded by u_6, u_{12} and $b = v_m$; in this case, u_6 itself is the apex.

In our analysis, the difference of two funnels turns out to be important. The idea is applicable to both U and V chains, but we define it here just for the V-chain. Given two vertices $v_i, v_j \in V$, where $i > j$, their funnel-difference is defined as

$$\Delta F(v_i, v_j) \;=\; F(v_i) \setminus F(v_j). \tag{5}$$

That is, $\Delta F(v_i, v_j)$ is the set of edges that are in $F(v_i)$ but not in $F(v_j)$. The edges of $\Delta F(v_i, v_j)$ form a contiguous portion of $F(v_i)$ surrounding v_i. The following property of the funnel-difference will be most important to us.

Lemma 3.3 *$\Delta F(v_i, v_j)$ is edge-disjoint from $F(v_k)$, for any $k < j < i$.*

> **Proof:** The proof uses the following elementary properties of the lowest common ancestor in an ordered tree: (i) the lowest common ancestor relation is commutative and associative, and (ii) the lowest common ancestor of two nodes x and y is above or equal to the lowest common ancestor of any nodes lying between x and y (in symmetric order). These properties imply that $\alpha(v_i, v_k)$ is above or equal to $\alpha(v_i, v_j)$ in T_a and $\beta(v_i, v_k)$ is above or equal to $\beta(v_i, v_j)$, for any $k < j < i$. Thus, no edge of $F(v_k)$ lies in $\Delta F(v_i, v_j)$. This completes the proof. $\Box$

3.3 Data structures for computing tangents on funnels

Our main operation on funnels is finding a common tangent between two funnels. Consider a vertex-pair $u \in U$ and $v \in V$. The funnel-pair $(F(u), F(v))$ is *closed* if $\pi(\alpha(u), \beta(u))$ and $\pi(\alpha(v), \beta(v))$ are disjoint except perhaps at endpoints, and *open* otherwise.

An open funnel-pair always admits a unique common tangent. If $(F(u), F(v))$ is an open funnel-pair, then the shortest path $\pi(u, v)$ necessarily uses the tangent between their funnels. We denote the tangent for the funnel-pair $(F(u), F(v))$ by $\ell(u, v)$. The endpoints of $\ell(u, v)$ are denoted u_ℓ and v_ℓ, where $u_\ell \in U$ and $v_\ell \in V$. (Throughout, we adopt the convention that the subscript ℓ will denote the endpoint of a tangent.)

In Figure 3, funnels $F(v_{10})$ and $F(u_6)$ form a closed pair, while $F(v_{10})$ and $F(u_{19})$ form an open pair. The tangent between $F(v_{10})$ and $F(u_{19})$ is (u_{22}, v_{15}).

During each execution of *REDUCE*, we maintain funnels $F(u)$ and $F(v)$ for the current vertices u and v. As observed in Section 2.3, u moves backward and forward among the active vertices of U, while v moves only forward through the active vertices of V. After each movement of u or v, we compute the tangent for $(F(u), F(v))$, if it is an open funnel-pair.

Let us focus on one of the shortest paths in $F(u)$, say, $\pi(z(u), a)$. We assume that the apex $z(u)$ of $F(u)$ is precomputed; precomputation of all funnel-apexes takes $O(n)$ time altogether using the two shortest path trees T_a and T_b.

A natural representation of $\pi(z(u), a)$ is a doubly linked list of vertices, but unfortunately this data structure is too inefficient for searching in our application. We speed up the search by breaking the path into subpaths, each represented by a *super-node* that supports fast searches. In this representation, a shortest path is a list of super-nodes, linked by *super-edges*. In phase k of the algorithm, the subpath within a single super-node has at most 2^k vertices. It is represented by a binary tree of height at most k. The leaves store the subpath vertices. Each internal node represents the subpath of its leaf descendants; it stores the edge linking its left and right subtrees' paths.

Our tangent-finding algorithm starts its search from two start vertices $u_s \in F(u)$ and $v_s \in F(v)$. The algorithm is a standard binary search [8, 14], with a modified search sequence. Instead of starting from the middle edge of the funnel and then recursively cutting the funnel in half, the search on $F(u)$ walks up from u_s to the root of the super-node binary tree containing u_s, follows the super-edges between super-nodes, and descends into the super-node binary tree containing the tangent endpoint u_ℓ. The search uses funnel edges in its comparisons, namely the super-edges it follows between super-nodes and the edges above u_s and u_ℓ in the super-node binary trees containing those vertices. See Figure 4. The following lemma bounds the time of the search.

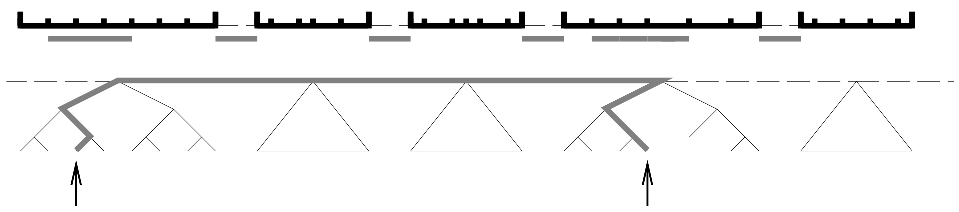

Figure 4: The super-node representation. The funnel path is shown schematically on top, and its super-node representation on the bottom. The positions of u_s and u_ℓ are shown by arrows, u_s on the left. The search path through the super-node representation is shaded gray, as are the funnel edges examined by the search.

Lemma 3.4 *Let $(F(u), F(v))$ be an open funnel-pair with a common tangent $\ell(u,v) = (u_\ell, v_\ell)$, with each funnel represented by two paths of super-nodes. Let n_u be the number of super-nodes on $F(u)$ between the start vertex u_s and u_ℓ for a particular tangent search; define n_v analogously. If the search occurs during the kth phase of MSEARCH, then computing the tangent $\ell(u,v)$ takes $O(n_u + n_v + k)$ time.*

We need to show how to maintain this representation of $F(u)$ (actually of $\pi(z(u), a)$) as u moves among the active vertices of U, and how to initialize the data structure at the start of each phase. To accomplish both tasks, we consider the shortest path tree T_a. In the kth phase we will retain only the vertices of T_a that are ancestors of at least one apex $z(u)$ for some active $u \in U$. This gives a tree with at most n_k leaves, and hence with at most $n_k - 1$ nodes of degree more than two. If we cut T_a below each node of degree greater than two and below each active apex, we get a decomposition of T_a into at most $2n_k - 1$ paths. We partition each of these paths into super-nodes, each with at most 2^k vertices. Linking the paths back together gives a shortest path tree of super-nodes. We will ensure that the total number of super-nodes in the tree is $O(kn_k) = O(kn2^{-k})$.

Given this tree of super-nodes and the path of super-nodes representing $\pi(z(u), a)$, it is relatively easy to produce the path representing $\pi(z(u'), a)$ for u' an active neighbor of u in U. We compute $\alpha(u, u')$ and map it to the super-node X containing it. The descendant of X is Y, a super-node ancestor of u but not of u'. One of the (at most two) immediate siblings of Y in the shortest path tree is Z, a super-node ancestor of u' but not of u. We unlink X from Y, and link X to Z instead. (If u' is an ancestor or descendant of u, the work required is different, but still easy.) All this takes only constant time.

In phase 0 the shortest path tree of super-nodes is just the original tree T_a: all vertices of U are active, and all super-nodes contain exactly one vertex. To produce the tree of super-nodes for phase $k+1$, given the tree for phase k, we delete dead vertices and combine neighboring super-nodes. More precisely, we identify the super-nodes that contain active apexes in phase $k+1$ and delete all super-nodes that are not (improper) ancestors of at least one such super-node. We cut the tree below every super-node of degree greater than two and every super-node containing an active apex. This gives a decomposition into at most $2n_{k+1}-1$ paths. On each path with at least two super-nodes, we merge adjacent pairs of super-nodes. "Merging" two super-nodes means that we build a new binary tree whose left and right subtrees are the old super-node trees, and whose root contains the super-edge linking the old super-nodes. It is not hard to show by induction on k that the number of super-nodes is at most $(\frac{k+1}{2})n_k$ for $k > 0$. We summarize our data structure's properties in the following lemma.

Lemma 3.5 *During phase k, the shortest path tree of super-nodes contains $O(kn_k)$ super-nodes. The overhead to prepare the data structure for phase k is $O(kn_k)$. Given the super-node representation of $F(u)$, we can modify it to obtain the representation of $F(u')$ for a neighboring active vertex u' in constant time.*

Remark: The bound on the number of super-nodes during the kth phase can be improved to $O(n_k)$, at the expense of increasing the tree height to $2k$. The $O(kn_k)$ bound results from merging each super-node tree with at most one other at each phase; this allows the tree height to remain at most k in phase k. By performing two rounds of merging per phase, we can ensure that the number of super-nodes is $O(n_k)$, but the two rounds of merging may increase the tree height to $2k$. We use one round of merging because the inferior super-node bound of $O(kn_k)$ is sufficient for our purpose.

3.4 Distance Computation

In this section, we show how the primitives of the previous two subsections are used to perform a matrix evaluation. Suppose we want to compute the distance $d(u,v)$, where $u \in U$ and $v \in V$. We start by computing the four lowest common ancestors $\alpha(u), \beta(u), \alpha(v), \beta(v)$. By performing three ancestor queries, we can determine the exact order of these four vertices along the path $\pi(a,b)$. The funnel-pair $(F(u), F(v))$ is closed if and only if the order is either $(\alpha(u), \beta(u), \alpha(v), \beta(v))$ or $(\alpha(v), \beta(v), \alpha(u), \beta(u))$. Otherwise, the pair is an open funnel-pair. Since each ancestor query takes $O(1)$ time, the time to determine whether the funnel-pair is open or not is only a constant. The following two lemmas detail how to compute the distance $d(u,v)$ depending upon whether the funnel-pair is open or closed.

Lemma 3.6 *If the funnel-pair $(F(u), F(v))$ is closed, $d(u,v)$ can be computed in $O(1)$ time.*

Proof: Without loss of generality, assume that the lowest common ancestor vertices $\alpha(v), \beta(v)$ precede $\alpha(u), \beta(u)$ on $\pi(a,b)$. See Figure 3 for illustration; the funnel-pair $(F(u_6), F(v_{10}))$ is closed. Then it is easy to see that

$$d(u,v) = d(u,\alpha(u)) + d(\alpha(u),\beta(v)) + d(v,\beta(v)). \tag{6}$$

These distances can be deduced from shortest path trees T_a and T_b in constant time each. For instance, $d(u,\alpha(u)) = d(a,u) - d(a,\alpha(u))$, and both the distances on the right hand side are precomputed in T_a. This completes the proof. □

Lemma 3.7 *If the funnel-pair $(F(u), F(v))$ is open, $d(u,v)$ can be computed in $O(\tau(u,v))$ time, where $\tau(u,v)$ is the time to compute the tangent $\ell(u,v)$.*

Proof: We compute the tangent $\ell(u,v)$ for the funnel-pair $(F(u), F(v))$. Refer to Figure 3, where the funnel-pair $(F(u_{19}), F(v_{10}))$ is open. Then it is easy to see that

$$d(u,v) \;=\; d(u,u_\ell) + d(u_\ell, v_\ell) + d(v_\ell, v). \tag{7}$$

The distance $d(u_\ell, v_\ell)$ is the Euclidean distance between u_ℓ and v_ℓ. The remaining two terms are deduced from the shortest path trees T_a and T_b in constant time. For example, consider the distance $d(u, u_\ell)$. If u_ℓ is an ancestor of u in T_a, then $d(u,u_\ell) = d(a,u) - d(a,u_\ell)$; otherwise u_ℓ is an ancestor of u in T_b, and $d(u,u_\ell) = d(b,u) - d(b,u_\ell)$. In either case, the distances to a and b are precomputed and accessible in constant time apiece. □

The preceding two lemmas have established that the dominant term in matrix operations is the cost of finding tangents between open funnel-pairs.The next two sections contain the main result of our paper. Section 4 presents our algorithm for computing tangents, and Section 5 analyzes the algorithm, showing that it takes $O(k)$ amortized time to compute a tangent in the kth phase of the matrix-searching.

4 The Algorithm

We describe how to perform the matrix operations during one phase of the subroutine *REDUCE*. Suppose that we are in the kth phase of the algorithm: the submatrix A_k has size $n_k \times 2n_k$, where $n_k \le n2^{-k}$. Matrix-searching theory tells us that *REDUCE* performs at most $3n_k$ matrix evaluations. We label the matrix evaluations in time-order, where time runs in integer steps from 1 to $3n_k$. By convention, the matrix entry being evaluated at time t is $d(u(t), v(t))$, where $1 \le t \le 3n_k$ and $u(t) \in U$ and $v(t) \in V$ are active vertices. The preceding section shows that the dominant cost in computing $d(u(t), v(t))$ is finding the tangent $\ell(u(t), v(t))$. Therefore, in the remainder of this section, we focus on the tangent-finding component of the matrix operations.

We use the shorthand notation $\ell(t)$ to denote the tangent computed at time t:

$$\ell(t) \;=\; \ell(u(t), v(t)).$$

The vertices $u_\ell(t)$ and $v_\ell(t)$, as before, denote the endpoints of $\ell(t)$. Our main subroutine for implementing *REDUCE* is called *ADVANCE*; it computes the tangent $\ell(t+1) = \ell(u(t+1), v(t+1))$ and advances the current time from t to $t+1$. We maintain the following two auxiliary data fields to support the operations in the subroutine at the current time t:

1. $\ell(t)$: This is the tangent computed at time t; that is, $\ell(t) = \ell(u(t), v(t))$. We store $u_\ell(t)$ and $v_\ell(t)$, the two endpoints of $\ell(t)$, at vertex $v(t)$.
2. $\tilde{\ell}(u)$: This is the most recent tangent computed at the vertex u prior to the current time t; the time t is implicit in this definition. Each vertex $u \in U$ maintains its $\tilde{\ell}(u)$, which is initially set to *NIL*.

The pseudocode for *ADVANCE* (below) finds the tangent $\ell(t+1)$ between the funnels $F(u(t+1))$ and $F(v(t+1))$. The algorithm has two nearly identical parts, one executed when the V-vertex advances, and the other when the U-vertex advances. The two parts differ in one important detail: the choice of the vertex where the search for new tangent begins. When the V-vertex advances (Step 1), the search starts at the lowest common ancestor $\alpha(v(t), v(t+1))$ in V, and at the previous tangent endpoint $u_\ell(t)$ in U. However, when the U-vertex advances (Step 2), a more involved choice of the start vertex is needed. We introduce these vertices, $u^*(t+1)$ and $v^*(t+1)$, in the following; see Figure 5.

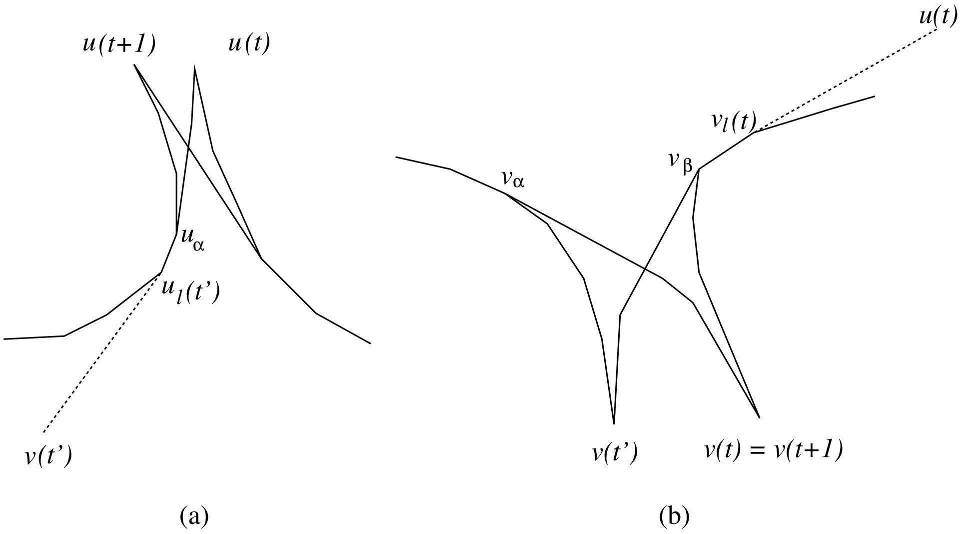

Figure 5: The definition of the start vertices $u^*(t+1)$ and $v^*(t+1)$. In Figure (a), $u^*(t+1) = u_\alpha$, and in Figure (b), $v^*(t+1) = v_\beta$.

Let t' denote the time when $\tilde{\ell}(u(t+1))$ was computed; it is undefined if $\tilde{\ell}(u(t+1))$ is *NIL*. Thus $\ell(t') = \tilde{\ell}(u(t+1))$. The two endpoints of $\ell(t')$ are denoted $u_\ell(t')$ and $v_\ell(t')$. Let $u_\alpha = \alpha(u(t), u(t+1))$. Then the start vertex $u^*(t+1)$ is defined as follows.

$$u^*(t+1) \;=\; \begin{cases} u_\alpha & \text{if } t' \text{ is undefined or if } u_\alpha \text{ is a descendant of } u_\ell(t') \text{ in } T_a, \\ u_\ell(t') & \text{otherwise.} \end{cases}$$

We observe that $u^*(t+1)$ is a vertex in the funnel-difference $\Delta F(u(t+1), u(t))$.

Next, we define $v^*(t+1)$, which is a vertex in the funnel-difference $\Delta F(v(t), v(t'))$; recall that $v(t')$ is the vertex on the V-chain that was current at time t'. We define v_α and v_β as the vertices bounding the funnel-difference $\Delta F(v(t), v(t'))$:

$$v_\alpha \;=\; \alpha(v(t), v(t')), \qquad v_\beta \;=\; \beta(v(t), v(t')).$$

Finally, recall that $v_\ell(t)$ is the V-chain endpoint of the tangent $\ell(t)$. The start vertex $v^*(t+1)$ is defined as follows.

$$v^*(t+1) = \begin{cases} v_\alpha & \text{if } t' \text{ is defined and } v_\alpha \text{ is a descendant of } v_\ell(t) \text{ in } T_a, \\ v_\beta & \text{if } t' \text{ is defined and } v_\beta \text{ is a descendant of } v_\ell(t) \text{ in } T_b, \\ v_\ell(t) & \text{otherwise.} \end{cases}$$

With these definitions, we are ready to present the pseudocode for *ADVANCE*. The pseudocode talks about "painting" edges, which is merely a device used by our analysis to prove the linearity of the algorithm. To perform the pseudocode test whether $\ell(t+1)$ is the same as a previously computed tangent ℓ', the algorithm checks whether ℓ' is still tangent to $F(u(t+1))$ and $F(v(t+1))$ at its endpoints. This takes constant time using lowest common ancestor queries.

ADVANCE(t)

(* *By matrix-searching theory, either* $v(t+1) \neq v(t)$ *or* $u(t+1) \neq u(t)$*, but not both.* *)

1. **if** $v(t+1) \neq v(t)$ **then**
   (a) **if** $\ell(t+1) = \ell(t)$ **then**
      Set $\ell(t+1) = \ell(t)$;
   (b) **else**
      Simultaneously search for $v_\ell(t+1)$ on $F(v(t+1))$ starting from $\alpha(v(t), v(t+1))$, and for $u_\ell(t+1)$ on $F(u(t+1))$ starting from $u_\ell(t)$.
      Paint the edges of $F(u(t+1))$ between $u_\ell(t)$ and $u_\ell(t+1)$;

2. **else** (* $u(t+1) \neq u(t)$ *)
   (a) **if** $\ell(t+1) \in \{\ell(t), \tilde{\ell}(u(t+1))\}$ **then**
      Set $\ell(t+1)$ accordingly;
   (b) **else**
      Simultaneously search for $v_\ell(t+1)$ on $F(v(t+1))$ starting from $v^*(t+1)$, and for $u_\ell(t+1)$ on $F(u(t+1))$ starting from $u^*(t+1)$.
      Paint the edges of $F(v(t+1))$ between $v^*(t+1)$ and $v_\ell(t+1)$, and paint the edges of $F(u(t+1))$ between $u^*(t+1)$ and $u_\ell(t+1)$;

end *ADVANCE*

ADVANCE correctly computes the new tangent $\ell(t+1)$: Steps 1a and 2a set $\ell(t+1)$ to a previously computed tangent if that tangent is still valid at time $t+1$. The remaining steps search for the tangent endpoints on funnels $F(u(t+1))$ and $F(v(t+1))$. Since $u_\ell(t+1) \in F(u(t+1))$ and $v_\ell(t+1) \in F(v(t+1))$, these searches are also correct. (The choice of the vertices where the search starts is relevant only for analyzing the running time.)

5 The Analysis

This section establishes the linearity of our matrix-searching algorithm. In the kth phase of *MSEARCH*, the submatrix input to *REDUCE* has size $n_k \times 2n_k$, where $n_k \leq n2^{-k}$, and the total number of matrix evaluations performed by *REDUCE* is $O(n_k)$. The discussion in Subsection 3.4 shows that the only part of a matrix operation requiring non-constant work is the tangent computation. Thus, the running time of *ADVANCE* dominates the running time of *REDUCE* in a phase.

5.1 The analysis of *ADVANCE*

The running time of *ADVANCE* is dominated by the searches performed in the tangent-finding steps. The remaining steps take only constant time apiece per step: the tests of tangent equivalence in Steps 1a and 2a can be made in constant time, and the start vertices $u^*(t+1)$ and $v^*(t+1)$ can also be computed in constant time.

We bound the cost of computing tangents in *ADVANCE* during the kth phase by arguing that each funnel edge is searched over at most twice. Using the terminology of Section 3.3, if u_s, v_s are the start vertices for the tangent search, then the edges of $F(u)$ between u_s and u_ℓ and the edges of $F(v)$ between v_s and v_ℓ are considered to be *searched over* when $\ell(u,v)$ is computed. (The key point is that the actual cost of computing the tangent is $O(n_u + n_v + k)$, where $n_u + n_v - 2$ is the number of super-edges searched over during this tangent-finding operation; see Lemma 3.4.)

Except for the search on the V-chain in Step 1b, all other searches in *ADVANCE* are accounted for by a "painting" paradigm: we paint the edges that are searched over during a tangent computation, and each edge is painted at most once (Lemmas 5.3 and 5.4). We use a separate accounting scheme (Lemma 5.1) to count the edges of V searched over in Step 1b. Together, these lemmas prove that no edge of either U or V is searched over more than twice during a complete phase of *REDUCE*.

Lemma 5.1 *During a phase of REDUCE, each edge in the V-chain is searched over in Step 1b of ADVANCE at most once.*

Proof: Step 1 is executed when $u(t+1) = u(t)$, and $v(t+1) \neq v(t)$. The search in Step 1b is carried out if and only if $\ell(t+1) \neq \ell(t)$. These two facts together imply that $v_\ell(t+1)$ necessarily lies in the funnel-difference $\Delta F(v(t+1), v(t))$. Only edges of the funnel-difference $\Delta F(v(t+1), v(t))$ are searched over. By Lemma 3.3 the funnel-differences are disjoint at any two distinct times: $\Delta F(v(t'+1), v(t')) \cap \Delta F(v(t''+1), v(t'')) = \emptyset$, for any $0 < t' < t'' < 3n_k$. This completes the proof. □

The proofs of the two remaining lemmas (Lemmas 5.3 and 5.4) are more complicated, and they rely on the following technical lemma.

Lemma 5.2 *Whenever edges of the V-chain are painted at time $t+1$, their extensions hit the U-chain between $u(t)$ and $u(t+1)$. Similarly, whenever edges of the U-chain are painted at time $t+1$, their extensions hit the V-chain before $v(t+1)$. If the edges of the U-chain are painted because $v(t+1) \neq v(t)$, then their extensions hit the V-chain between $v(t)$ and $v(t+1)$.*

Proof: The proof of the lemma depends on the fact that the painted edges lie on a funnel, and the edges of a funnel have either monotonically decreasing or monotonically increasing slopes from one end to the other. The edges of V painted at time $t+1$ are in $F(v(t+1))$, and they lie between $v^*(t+1)$ and $v_\ell(t+1)$. Because $v_\ell(t+1)$ belongs to $\Delta F(v(t+1), v(t'))$, $v^*(t+1)$ lies on $F(v(t+1))$ between $v_\ell(t)$ and $v_\ell(t+1)$. By the slope property, the slopes of all the painted edges are between the slopes of $\ell(t)$ and $\ell(t+1)$. By the regular expression of Section 2.3, $u(t)$ and $u(t+1)$ are two adjacent active vertices in U; notice that $u(t+1)$ can be either the successor or the predecessor of $u(t)$, but the two vertices must be adjacent. Because the extensions of all the painted edges lie between $\pi(v(t), u(t))$ and $\pi(v(t), u(t+1))$, they must hit the U-chain between $u(t)$ and $u(t+1)$. Figure 6(a) illustrates this case: the edges painted are shown with heavy lines.

The edges of U can be painted either in Step 1b or in Step 2b. If the edges are painted in Step 1b, then $v(t+1) \neq v(t)$, but $u(t) = u(t+1)$. In this case, the painted edges lie between $u_\ell(t)$ and $u_\ell(t+1)$, the U-endpoints of the tangents at times t and $t+1$. Again, by the slope property, all painted edges have slopes in the range determined by the slopes of $\ell(t)$ and $\ell(t+1)$. Since $v(t)$ and $v(t+1)$ are adjacent active vertices of V, the extensions of all painted edges hit the V-chain between $v(t)$ and $v(t+1)$.

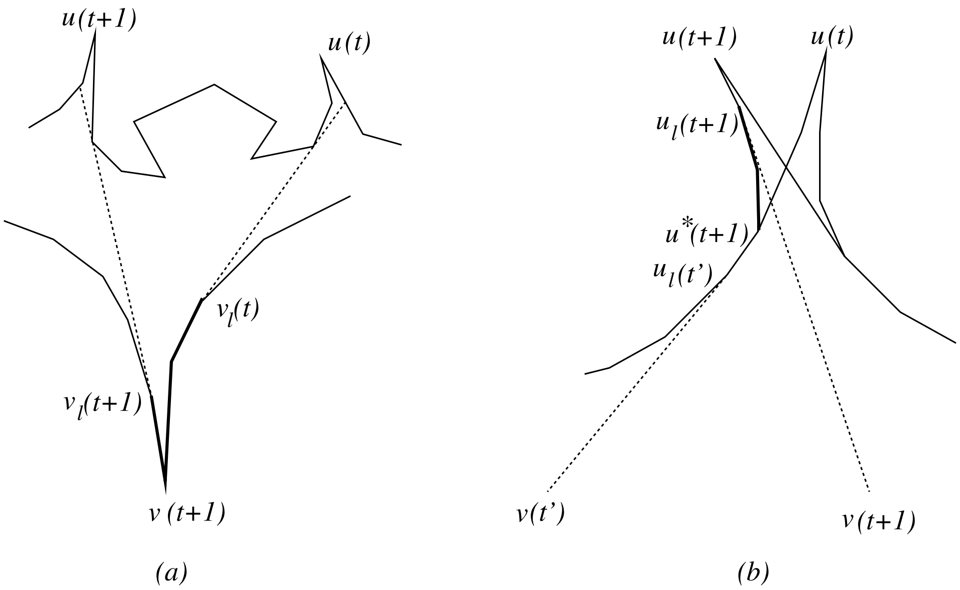

Figure 6: Illustration for the proof of Lemma 5.2. Figure (a) shows the painting of V-edges; a symmetric case arises when $u(t)$ is after $u(t+1)$. Figure (b) shows the painting of U-edges (in Step 2b); in this example, the start vertex is $u^*(t+1) = \alpha(u(t), u(t+1))$, and t' is the time when the old tangent $\tilde{\ell}(u(t+1))$ was computed.

Finally, if the U-edges are painted in Step 2b, then $v(t) = v(t+1)$, but $u(t) \neq u(t+1)$. In this case, the painted edges lie between $u^*(t+1)$ and $u_\ell(t+1)$. Since the matrix-searching algorithm never backs up on the V-chain, the tangent endpoint on $F(u(t+1))$ moves monotonically clockwise with time. In particular, if the old tangent $\tilde{\ell}(u(t+1))$ is not *NIL*, it lies on the shortest path to some vertex $v(t')$ that is before

$v(t+1)$ in the V-chain order. Because $u_\ell(t+1)$ belongs to $\Delta F(u(t+1), u(t))$, $u^*(t+1)$ lies on $F(u(t+1))$ between $u_\ell(t')$ and $u_\ell(t+1)$. If $\tilde{\ell}(u(t+1))$ is *NIL*, then $u^*(t+1) = \alpha(u(t), u(t+1))$ is counterclockwise of $u_\ell(t+1)$. In either case, the slope property ensures that the extensions of the U-edges painted in Step 2b hit the V-chain before $v(t+1)$. Figure 6(b) illustrates this case: the edges painted are shown with heavy lines. This completes the proof. $\square$

The next two lemmas show that no edge of U or V is painted more than once in a phase. Both lemmas rely on the following observations about Step 2b of *ADVANCE*, in which $v(t+1) = v(t)$ and $u(t+1) \neq u(t)$, and the tangent $\ell(t+1)$ equals neither $\ell(t)$ nor $\tilde{\ell}(u(t+1))$. Let t' denote the time when $\tilde{\ell}(u(t+1))$ is computed, assuming it is not *NIL*; that is, t' is the most recent time before t when a tangent is computed for $F(u(t+1))$. We observe the following vertex-relations:

$$u(t') = u(t+1), \qquad u(t'+1) = u(t), \qquad v(t') = v(t'+1). \tag{8}$$

The first relation follows from the definition of t'. The third follows from the first: at time $t'+1$, the algorithm must have advanced the U-vertex from $u(t')$ to $u(t'+1)$, keeping the V-vertex the same, since assuming otherwise contradicts the choice of time t'. Finally, the second relation follows from the important fact that *ADVANCE* visits the vertices of U in chain order (either forward or backward), never skipping an active vertex. The direction of movement in the U-chain at time t is opposite to the direction at time t'. A key consequence of the vertex relations in Eq. (8) is that between times t' and t, the algorithm visits vertices that are all separated from $u(t+1)$ by $u(t)$. That is, $u(t)$ separates $u(t+1)$ from $u(t'')$, for $t'+1 < t'' < t$.

Lemma 5.3 *An edge of the V-chain is painted at most once per phase.*

Proof: The edges of V are painted only in Step 2b. Our analysis makes crucial use of the funnel-difference $\Delta F(v(t), v(t'))$. Since $\ell(t+1) \neq \ell(t')$, it follows that $v_\ell(t+1) \in \Delta F(v(t), v(t'))$. We break up our analysis in two cases, depending upon whether or not $v_\ell(t) \in \Delta F(v(t), v(t'))$.

Case 1. [$v_\ell(t) \in \Delta F(v(t), v(t'))$, or t' is undefined]. See Figure 7.

In this case, we observe that $v^*(t+1) = v_\ell(t)$. Since $v_\ell(t+1)$ lies in the funnel-difference $\Delta F(v(t), v(t'))$, all edges of V painted at time $t+1$ belong to $\Delta F(v(t), v(t'))$. By Lemma 5.2, the extensions of the newly painted edges hit the U-chain between $u(t)$ and $u(t+1)$. If t' is undefined, no painted edge's extension has hit here before, so the newly painted edges have never been painted before. If t' is defined, the most recent time *ADVANCE* traversed this portion of U was at time $t'+1$, when the U-vertex moved from $u(t')$ to $u(t'+1)$. Since the V-funnel at time $t'+1$, namely $F(v(t'))$, is edge-disjoint from $\Delta F(v(t), v(t'))$, the newly painted edges were not painted at time step $t'+1$. Because $u(t)$ separates $u(t+1)$ from $u(t'')$ for any $t'+1 < t'' < t$, it follows from Lemma 5.2 that the edges were not painted between time $t'+1$ and time t. By Lemma 3.3, these edges were not painted at any time $t'' \leq t'$, since $\Delta F(v(t), v(t'))$ is disjoint from $F(v(t''))$. Thus, the edges of V painted at time $t+1$ have never been painted before.

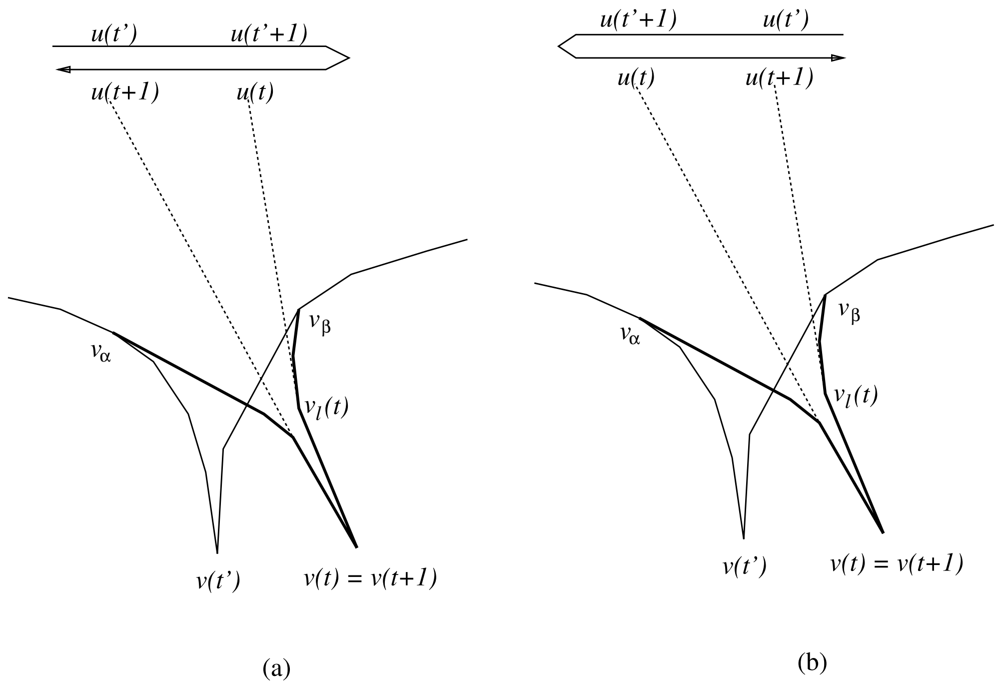

Figure 7: Illustration for Case 1 of Lemma 5.3. Figures (a) and (b) show the two symmetric cases: in (a), the algorithm is moving forward on the U-chain at time t, and in (b), it is moving backward at time t. The motion is suggested by the arrow on top. The funnel-difference $\Delta F(v(t), v(t'))$ is shown in heavy lines.

Case 2. $[v_\ell(t) \notin \Delta F(v(t), v(t'))]$. See Figure 8.

Because $v_\ell(t) \in F(v(t'))$, we have $v_\ell(t) = v_\ell(t'+1)$. Furthermore, $v^*(t+1)$ equals either v_α or v_β, where $v_\alpha = \alpha(v(t), v(t'))$ and $v_\beta = \beta(v(t), v(t'))$. As noted above, $u(t)$ separates $u(t+1)$ from $u(t'')$, for $t' < t'' < t$. Since the V-vertex is confined between $v(t')$ and $v(t)$ during this time interval, we can be sure that tangent endpoints on V lie outside the funnel-difference $\Delta F(v(t), v(t'))$ between times t' and t. In particular, if $v^*(t+1) = v_\beta$, then $v_\ell(t'')$ lies at or above $v_\ell(t)$ in T_b, and if $v^*(t+1) = v_\alpha$, then $v_\ell(t'')$ lies at or above $v_\ell(t)$ in T_a, for any $t' < t'' < t$. Thus, any edges painted on V between times $t'+1$ and t are disjoint from $\Delta F(v(t), v(t'))$. Finally, no edge of $\Delta F(v(t), v(t'))$ is painted at or before t', since Lemma 3.3 ensures that $\Delta F(v(t), v(t'))$ is disjoint from $F(v(t''))$, for any $t'' \leq t'$. This completes the proof. $\square$

Lemma 5.4 *An edge of the U-chain is painted at most once per phase.*

Proof: The edges of U are painted either in Step 1b or in Step 2b. If the edges are painted in Step 1b, then $v(t+1) \neq v(t)$ and the painted edges lie between $u_\ell(t)$ and $u_\ell(t+1)$. By Lemma 5.2, the extensions of the newly painted edges hit the V chain between $v(t)$ and $v(t+1)$. Because $v(t)$ separates $v(t+1)$ from $v(t')$, for any $t' < t$, the newly painted edges cannot have been painted at an earlier time.

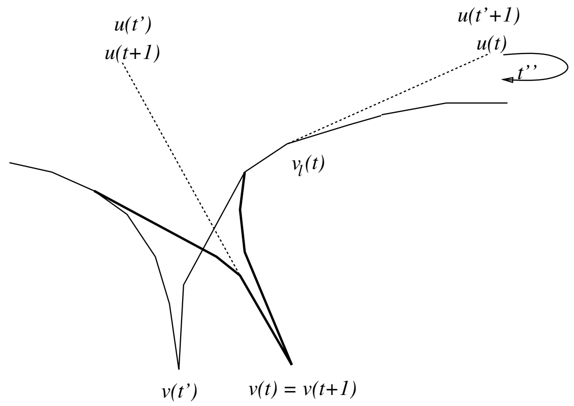

Figure 8: Illustration for Case 2 of Lemma 5.3. The symmetric case, in which the algorithm is moving backward on the U-chain at time t, is not shown.

If the edges are painted in Step 2b, then $u(t+1) \neq u(t)$, and the painted edges lie between the start vertex $u^*(t+1)$ and $u_\ell(t+1)$. Let t' denote the time when $\tilde{\ell}(u(t+1))$ is computed (possibly undefined). Since $u_\ell(t+1)$ equals neither $u_\ell(t)$ nor $u_\ell(t')$, we know that $u_\ell(t+1)$ lies in the funnel-difference $\Delta F(u(t+1), u(t))$. Since the start vertex $u^*(t+1)$ is guaranteed to be in $\Delta F(u(t+1), u(t))$, all the newly painted edges belong to $\Delta F(u(t+1), u(t))$. If t' is undefined, then by Lemma 3.3 no edge of $\Delta F(u(t+1), u(t))$ has appeared in $F(u(t''))$, for $t'' \leq t$, so none has been painted before. If t' is defined, the edges lying between $u_\ell(t')$ and $u_\ell(t+1)$ have not been painted at or before time t', since their extensions hit the V-chain after $v(t')$ (cf. Lemma 5.2). No edge of the funnel-difference $\Delta F(u(t+1), u(t))$ is painted between times $t'+1$ and t, since the lowest common ancestor property implies that $\Delta F(u(t+1), u(t))$ is disjoint from $F(u(t''))$, for any $t' < t'' \leq t$. Thus, none of the newly painted edges has been painted at an earlier time. This completes the proof. $\Box$

5.2 The analysis of *REDUCE* and *MFILL*

We show that the kth phase of *MSEARCH* takes $O(kn_k)$ time, where $n_k \leq n2^{-k}$. The analysis of the preceding section has shown that no edge of U or V is searched over more than twice. There are at most $3n_k$ matrix evaluations during *REDUCE* in the kth phase, and by Lemma 3.4 the time for one evaluation is $O(k + n_u + n_v)$, where $n_u + n_v - 2$ is the number of super-edges searched over during the evaluation. The total number of super-edges and the total overhead cost of maintaining the funnel data structure during the kth phase is $O((k+1)n_k)$. Thus, the total time spent on matrix evaluations in the kth phase of *REDUCE* is $O((k+1)n_k)$.

The running time of *MFILL* has the same bound. Section 2.3 shows that the matrix evaluations in *MFILL* satisfy the same regular expression as those in *REDUCE*. The same analysis applies, therefore, proving that the running time of *MFILL* at the kth phase is also $O((k+1)n_k)$.

5.3 The time complexity of *MSEARCH*

Finally, we are ready to show that our matrix-searching algorithm runs in $O(n)$ time overall. The algorithm consists of $O(\log n)$ phases, and the matrix in the kth phase has size at most $n_k \times 2n_k$, where $n_k \leq n2^{-k}$. In each phase, the running time is dominated by the subroutines *REDUCE* and *MFILL*; other steps take $O(n_k)$ time. The cost of executing *REDUCE* and *MFILL* in the kth phase is $O((k+1)n_k)$. Thus, the total cost of the matrix-searching algorithm *MSEARCH* is

$$T(n) = O\Big(\sum_{k=0}^{\lceil \log n \rceil} (k+1)n_k \Big) = O\Big(\sum_{k=0}^{\infty} k \frac{n}{2^k} \Big) = O(n).$$

We have established our main theorem.

Theorem 5.5 *Let P be a simple polygon with n vertices, and let U and V be subchains of its boundary that together partition the vertices of P. Let M denote a $p \times m$ matrix with $M(i,j)$ equal to the shortest path distance between the vertices $u_i \in U$ and $v_j \in V$, where $p = |U|$ and $m = |V|$. Then we can compute the maximum entry in each row of M in $O(n)$ total time.*

5.4 Computing row-wise minima

The algorithm of Section 4 also works for computing row-wise minima of M. In fact, by changing the ordering of columns and negating all distances, we can directly apply our maxima-finding algorithm to compute the minima. Let us define an auxiliary $p \times m$ matrix M', where

$$M'(i,j) \;=\; -M(i, m-j+1). \tag{9}$$

Notice that the column-ordering of M' is reverse of the column-ordering of M. The triangle inequality again implies that M' is totally monotone.

Lemma 5.6 *The matrix M' defined in Eq. (9) is totally monotone.*

We can find the row-wise maxima of M' (which correspond to the row-wise minima in M) using the matrix-searching algorithm of Section 4. The direction of travel in the V-chain is now reversed, due to the reversal of columns. To account for this change, we set $a = v_m$ and $b = v_1$. The algorithm and its analysis remain the same. We have the following theorem.

Theorem 5.7 *Let P be a simple polygon with n vertices, and let U and V be subchains of its boundary that together partition the vertices of P. Let M denote a $p \times m$ matrix with $M(i,j)$ equal to the shortest path distance between the vertices $u_i \in U$ and $v_j \in V$, where $p = |U|$ and $m = |V|$. Then we can compute the minimum entry in each row of M in $O(n)$ total time.*

6 Applications

In this section, we give several applications of Theorems 5.5 and 5.7. Our applications concern problems in simple polygons, where the shortest path metric is the most natural distance function. In addition, in several problems, we can use the shortest path metric to mask visibility constraints. Throughout this section, P denotes a simple polygon, whose vertices are labeled $p_1, p_2, \ldots, p_n$ in counterclockwise order around the boundary.

6.1 The geodesic diameter and farthest neighbors

This section describes a linear-time algorithm for computing a farthest neighbor for each vertex of P under the shortest path metric; the previous best time bound for this problem was $O(n \log n)$ [6, 16]. The *geodesic diameter* of P, denoted $D(P)$, is the maximum length of a shortest path in P:

$$D(P) \; = \; \max\{d(x,y) \mid x, y \in P\}.$$

The diameter is always realized by two vertices of P [16], and so finding it is a special case of the problem of computing farthest neighbors for all vertices of P. By computing a farthest neighbor of a vertex $p \in P$, we mean two things: identifying a vertex $q \in P$ such that $d(p,q) = \max\{d(p,p') \mid p' \in P\}$, and computing the distance $d(p,q)$. We use a lemma from [16] to decompose the all-farthest neighbor problem into at most three instances of maxima-finding in totally monotone matrices.

Let p_i, p_j and p_k be three vertices of P such that p_j is a farthest neighbor of p_i, and p_k is a farthest neighbor of p_j: $d(p_i,p_j) = \max\{d(p_i,p_l) \mid p_l \in P\}$, and $d(p_j,p_k) = \max\{d(p_j,p_l) \mid p_l \in P\}$. Without loss of generality, assume that p_i, p_j, p_k are in counterclockwise order. The triple p_i, p_j, p_k divides the polygon into three disjoint chains: $P_1 = (p_{i+1}, p_{i+2}, \ldots, p_{j-1})$, $P_2 = (p_{j+1}, p_{j+2}, \ldots, p_{k-1})$, and $P_3 = (p_{k+1}, p_{k+2}, \ldots, p_{i-1})$. Corresponding to these chains, are the three complementary chains: $Q_1 = (p_j, p_{j+1}, \ldots, p_i)$, $Q_2 = (p_k, p_{k+1}, \ldots, p_j)$, and $Q_3 = (p_i, p_{i+1}, \ldots, p_k)$. Figure 9 shows a schematic diagram of this construction. The following lemma is established in [16].

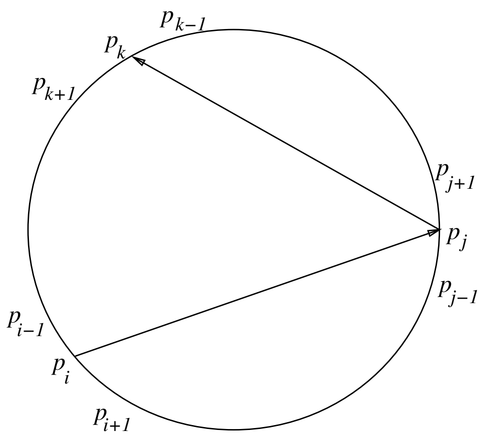

Figure 9: A schematic diagram showing chains P_i, for $i = 1, 2, 3$, in Lemma 6.1.

Lemma 6.1 ([16]) *Every vertex of the chain P_i has a farthest neighbor in chain Q_i, for $i = 1, 2, 3$.*

Since the farthest neighbors p_j and p_k can be computed in linear time, using shortest path trees [7, 11], Lemma 6.1 gives a linear-time reduction from the farthest neighbors problem of P to three instances of the farthest neighbors problem between disjoint chains. The chains P_i, for $i = 1, 2, 3$, cover all vertices of P except p_i, p_j and p_k. Since the farthest neighbors of p_i and p_j are already computed, only p_k remains, and we can compute its farthest neighbor in linear additional time. The farthest neighbors problem for the chains P_i and Q_i falls within the framework of Section 4, and is therefore equivalent to the problem of computing row-wise maxima in a totally monotone matrix. By Theorem 5.5, this can be done in $O(n)$ time, and we have the following theorem.

Theorem 6.2 *Given a simple polygon P on n vertices, we can compute a geodesic farthest neighbor for each of its vertices in $O(n)$ total time. Consequently, the geodesic diameter of P can also be computed in linear time.*

6.2 The external farthest neighbors

Agarwal et al. [1] gave an $O(n \log n)$ time algorithm for computing an external farthest neighbor for all vertices of a simple polygon. (The external farthest neighbors are computed using the *external* shortest path metric: the distance between two points is the length of a shortest path in the plane minus the interior of the polygon.) The bottleneck in their algorithm is the computation of *internal* farthest neighbors; the (internal) farthest neighbors algorithm is needed to compute the external farthest neighbors within a "pocket" of the polygon. The remaining steps of the external farthest neighbors algorithm take only $O(n)$ time. Thus, the following result is a direct corollary of Theorem 6.2.

Theorem 6.3 *Given a simple polygon P on n vertices, we can compute a geodesic external farthest neighbor for each of its vertices in $O(n)$ time. Consequently, the geodesic external diameter of P can also be computed in linear time.*

6.3 Weighted nearest neighbors and Euclidean matching

An internal diagonal of P divides the polygon into two chains. Let U and V be two polygonal chains obtained in this way, where $p = |U|$ and $m = |V|$. Let $w : P \to \Re$ be a real-valued weight function on the vertices of P, and define a $p \times m$ matrix M of weighted distances:

$$M(i, j) \;=\; d(u_i, v_j) - w(u_i) - w(v_j); \tag{10}$$

It is easily seen that M is totally monotone, by the triangle inequality. The problem of computing a *weighted nearest neighbor* for each vertex of U in V is equivalent to computing row-wise minima of M. This problem arises in an algorithm of Marcotte and Suri [13] for solving the Euclidean matching problem for points on a polygon. That algorithm finds an optimal matching for the vertices of a simple polygon, under the constraint that each matching edge lies inside the polygon. The algorithm is based on divide-and-conquer, and the dominating term in the conquer step is the complexity of solving the weighted nearest neighbors problem as defined above. Our algorithm of Section 4 solves the row-minima problem for M in linear time, thus improving the result in [13] from $O(n \log^2 n)$ to $O(n \log n)$.

Theorem 6.4 *The algorithm of Marcotte and Suri [13] for computing an optimal matching of the vertices of a simple polygon can be implemented in $O(n \log n)$ time, where n denotes the number of vertices of the polygon.*

6.4 Closest visible vertex pair between two chains

Given two disjoint polygonal chains U and V, the closest visible vertex problem for U and V is to find a closest pair of vertices $u \in U$ and $v \in V$ that are visible to each other. The visibility constraint implies that some vertex-pairs are not legal candidates, and thus the associated distance matrix is not fully defined. Aggarwal et al. [3] gave a complicated linear-time algorithm for this problem that exploits the special structure of the legal entries in the matrix. A linear-time algorithm due to Amato [4] corrects minor problems with the algorithm of Aggarwal et al. Amato's algorithm can also be implemented on a CREW PRAM in time $O(\log n)$ using $O(n)$ processors. However, like the algorithm of Aggarwal et al., Amato's algorithm is quite complicated. In this section, we show that a simpler and considerably more direct linear-time algorithm follows from our paradigm of matrix-searching using the shortest path metric.

Let M denote the $p \times m$ matrix whose (i, j) entry denotes the shortest path distance between u_i and v_j:

$$M(i, j) = d(u_i, v_j).$$

The triangle inequality implies that M is totally monotone, and so by Theorem 5.7, we can compute its row-minima in $O(n)$ time. Suppose that $M(i^*, j^*)$ is the overall minimum entry of M; clearly, this entry is also minimum in row i^*. We claim that the pair (u_{i^*}, v_{j^*}) is a closest visible pair in $U \times V$. By assumption, the distance between u_{i^*} and v_{j^*} is minimum over all u, v pairs, where $u \in U$ and $v \in V$; we need to show only that u_{i^*} and v_{j^*} are mutually visible. We assume otherwise, and arrive at a contradiction. If u_{i^*} and v_{j^*} are not visible, then the shortest path $\pi(u_{i^*}, v_{j^*})$ has at least one intermediate vertex, say, z. If $z \in U$, then $d(z, v_{j^*}) < d(u_{i^*}, v_{j^*})$, and if $z \in V$, then $d(u_{i^*}, z) < d(u_{i^*}, v_{j^*})$. In either case, the minimality of $M(i^*, j^*) = d(u_{i^*}, v_{j^*})$ is contradicted. Thus we have the following theorem.

Theorem 6.5 *Let U and V be two vertex-disjoint chains that partition the vertices of a simple polygon P. Then a closest visible vertex pair of U and V can be found in linear time.*

Theorem 6.5 applies equally well to computing the closest visible pair of vertices in two polygons. In linear time, we can easily transform the problem to two chains U and V that satisfy the conditions of the theorem.

7 Closing Remarks

We have presented a linear-time algorithm for computing the row-wise maxima or minima of a totally monotone matrix whose entries correspond to shortest path distances between two chains of a simple polygon. The result appears to be quite powerful and has several applications in computational geometry. The most direct applications concern distance-related problems in simple polygons, since the shortest path metric is the natural metric for nonconvex simple polygons. However, as Theorem 6.5 demonstrates, the result also has applications when the distances are Euclidean but there is an additional visibility constraint. The generalization to the shortest path metric helps mask the visibility constraint. The following table summarizes the main applications of our result.

Problem	**Previous Result**	**New Result**
Geodesic Diameter	$O(n \log n)$	$O(n)$
All Farthest Neighbors	$O(n \log n)$	$O(n)$
External Farthest Neighbors	$O(n \log n)$	$O(n)$
Euclidean Matching	$O(n \log^2 n)$	$O(n \log n)$
2-Chain Closest Visible Pair	$O(n)$	$O(n)$

Using some additional techniques in combination with the matrix-searching result of this paper, we have also obtained a linear-time algorithm for computing a shortest diagonal of a simple polygon. That result will be reported elsewhere [12].

References

[1] P. K. Agarwal, A. Aggarwal, B. Aronov, S. R. Kosaraju, B. Schieber, and S. Suri. Computing external-furthest neighbors for a simple polygon. *Discrete Applied Mathematics*, 31:97–111, 1991.

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

[3] A. Aggarwal, S. Moran, P. Shor, and S. Suri. Computing the minimum visible vertex distance between two polygons. In *Proc. of 1st Workshop on Algorithms and Data Structures*, pages 115–134. Springer-Verlag, 1989.

[4] N. Amato. Computing the minimum visible vertex distance between two nonintersecting simple polygons. In *Proceedings of the Conference on Information Sciences and Systems*, volume II, pages 800–805, Princeton, NJ, 1992.

[5] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6:485–524, 1991.

[6] L. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *Journal of Computer and System Sciences*, 39(2):126–152, 1989.

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

[8] L. Guibas, J. Hershberger, and J. Snoeyink. Compact interval trees: A data structure for convex hulls. *International Journal of Computational Geometry & Applications*, 1(1):1–22, 1991.

[9] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[10] J. Hershberger. A new data structure for shortest path queries in a simple polygon. *Information Processing Letters*, 38:231–235, 1991.

[11] J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. In *Proceedings of the 2nd Workshop on Algorithms and Data Structures*, pages 331–342. Springer-Verlag, 1991. Lecture Notes in Computer Science 519.

[12] J. Hershberger and S. Suri. Finding a shortest diagonal of a simple polygon in linear time. Manuscript, 1993.

[13] O. Marcotte and S. Suri. Fast matching algorithms for points on a polygon. *SIAM Journal on Computing*, 20:405–422, 1991.

[14] M. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[15] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. In *Proceedings of the Third Aegean Workshop on Computing*, pages 111–123. Springer-Verlag, 1988. Lecture Notes in Computer Science 319.

[16] S. Suri. Computing geodesic furthest neighbors in simple polygons. *Journal of Computer and System Sciences*, 39:220–235, 1989.