CONSTANT-WORK-SPACE ALGORITHMS FOR GEOMETRIC PROBLEMS*

Tetsuo Asano,† *Wolfgang Mulzer,*‡ *Günter Rote,*§ *and Yajun Wang*¶

Abstract. Constant-work-space algorithms may use only constantly many cells of storage in addition to their input, which is provided as a read-only array. We show how to construct several geometric structures efficiently in the constant-work-space model. Traditional algorithms process the input into a suitable data structure (like a doubly-connected edge list) that allows efficient traversal of the structure at hand. In the constant-work-space setting, however, we cannot afford to do this. Instead, we provide operations that compute the desired features on the fly by accessing the input with no extra space. The whole geometric structure can be obtained by using these operations to enumerate all the features. Of course, we must pay for the space savings by slower running times. While the standard data structure allows us to implement traversal operations in constant time, our schemes typically take linear time to read the input data in each step.

We begin with two simple problems: triangulating a planar point set and finding the trapezoidal decomposition of a simple polygon. In both cases adjacent features can be enumerated in linear time per step, resulting in total quadratic running time to output the whole structure. Actually, we show that the former result carries over to the Delaunay triangulation, and hence the Voronoi diagram. This also means that we can compute the largest empty circle of a planar point set in quadratic time and constant work-space. As another application, we demonstrate how to enumerate the features of an Euclidean minimum spanning tree (EMST) in quadratic time per step, so that the whole EMST can be found in cubic time using constant work-space.

Finally, we describe how to compute a shortest geodesic path between two points in a simple polygon. Although the shortest path problem in general graphs is NL-complete [27], this constrained problem can be solved in quadratic time using only constant work-space.

1 Introduction

Problem Setting and Motivation. The recent past has seen an explosive growth in storage capacity. With hard-drives surpassing the terabyte mark, programmers can exploit a virtually unlimited amount of work-storage for their programs. Alas, not infrequently this leads to space-inefficient programs that

*Preliminary versions of this work appeared as T. Asano and G. Rote. Constant-working-space algorithms for geometric problems. *Proc. 21st Canad. Conf. Comput. Geom. (CCCG)*, pp. 87–90, 2009; and T. Asano, W. Mulzer, and Y. Wang. Constant-work-space algorithm for a shortest path in a simple polygon. *Proc. 4th Workshop on Algorithms and Computation (WALCOM)*, pp 9–20, 2010.

†*School of Information Science, JAIST, Japan*, t-asano@jaist.ac.jp

‡*Institut für Informatik, Freie Universität Berlin, Germany*, mulzer@inf.fu-berlin.de

§*Institut für Informatik, Freie Universität Berlin, Germany*, rote@inf.fu-berlin.de

¶*Microsoft Research Asia, Beijing, China*, yajunw@microsoft.com

use too much storage and become too slow if sufficiently large memory is not available. Thus, we believe that space-efficient algorithms deserve more attention.

Not only do such algorithms provide a counter-balance to the wastefulness of some contemporary software, but they become indispensable for built-in or embedded software in highly functional hardware, such as digital cameras and scanners. Sensor networks provide an excellent example: with flash memory becoming cheaper, even a large number of inexpensive sensors can be equipped with huge-volume flash drives. While the data measured by the sensor must be stored onboard for processing and needs to be written, it is preferable to write to the flash drive as little as possible, since this is a slow and expensive operation and reduces the flash drive's lifetime. Hence, we would like to process the data while performing only read operations on the flash drive and using only higher level memory for writing (e.g., CPU registers). We measure an algorithm's space efficiency by the number of work storage cells it uses. Ultimate space efficiency is achieved by a *constant-work-space algorithm*, i.e., an algorithm that uses only a constant number of variables in addition to the input storage. Such algorithms are also said to run in "log-space", since the amount of work-space is $O(\log n)$ bits for input size n [1].

Our Results. In this paper we present constant-work-space algorithms for several fundamental geometric problems, including constructing a triangulation for a planar point set; finding the trapezoidal decomposition of a polygonal region; computing the Delaunay triangulation, Voronoi diagram and Euclidean MST of a planar point set; and finding a geodesic shortest path in a simple polygon.

Traditional algorithms for finding these structures proceed by constructing a suitable data structure (typically a doubly-connected edge list, DCEL, or a similar structure [6, 24]) that allows to traverse and manipulate the features of the structure in question. These operations usually consist of finding the clockwise next edge for a given edge incident to one of its endpoints; finding the triangles or trapezoids incident to a given edge; finding the twin edge for a given edge; etc. With the DCEL at hand, these operations can be carried out in constant time. In our setting, however, we do not have the space for such a structure at our disposal. Instead, we establish a scheme for executing these operations without referring to any data structures. Usually, each operation needs a single scan over the input data, resulting in linear time per operation, and quadratic time overall.

We begin with algorithms for computing a triangulation of a planar point set and for finding the decomposition of a simple polygon into trapezoids. Both are classic problems in computational geometry, and they can be solved traditionally in time $O(n \log n)$ [6, 24] and $O(n)$ [11], respectively. We give algorithms that enumerate all features of the triangulation and the trapezoidation in quadratic time. It is easy but a bit tedious to adapt our algorithms for such a scheme as described above, and we omit these details.

Instead, we showcase the details of such an algorithm for the Delaunay triangulation of a planar n-point set. Several traditional $O(n \log n)$-time algorithms for constructing Delaunay triangulations are known [6]. As described above, we provide efficient algorithms for supporting the operations of a DCEL on the fly. Each operation takes a single scan over the point set and hence needs linear time. Thus, we can enumerate all the features of the Delaunay triangulation in quadratic time and with constant work-space. Using these operations, we can also solve related problems in constant work-space, such as finding the largest empty disc of n points in $O(n^2)$ time and enumerating the edges of the planar Euclidean Minimum Spanning tree of n points in $O(n^3)$ time. It is now also easy to support similar operations for the Voronoi diagram of a planar point set. For example, we can follow the boundary of

a given Voronoi region and find the clockwise next edge incident to a given Voronoi vertex in linear time. Hence, we can draw the Voronoi diagram for a planar point set in quadratic time.

Finally, we address the problem of finding a shortest geodesic path between two points in a simple polygon. We present an efficient algorithm using geometric properties which runs in $O(n^2)$ time for a simple polygon with n vertices. This algorithm is much simpler than our previous solution [4].

Our algorithms need more time than those in the standard computational model, but when considering the product of the time and space requirements, we actually often improve over the previous results: while existing algorithms for Delaunay triangulations and Voronoi diagrams need $O(n \log n)$ time and linear space, resulting in a time-space product of $O(n^2 \log n)$, we obtain a time-space product of $O(n^2) \times O(1) = O(n^2)$.

Related Results. Constant-work-space algorithms have been studied in complexity theory under a different name, "log-space" algorithms [1]. Notwithstanding, the authors prefer the current name, since it is more intuitive. There have been several previous results on log-space algorithms. One of the most important of these results is the selection algorithm by Munro and Raman [23] which runs in $O(n^{1+\epsilon})$ time using work-space $O(1/\epsilon)$, for any small constant $\epsilon > 0$. In 2005, Reingold solved a long-standing open problem in complexity theory by describing a deterministic log-space algorithm for finding a path between two given vertices in an undirected graph [25]. Asano [2,3] gives applications to image processing. Furthermore, there is a large number of algorithms for traversing and enumerating the vertices and facets of a given geometric structure (usually provided as a DCEL or a similar structure) without using any mark bits or recursion stacks [5, 7–9, 13, 15–17, 26].

A similar but more restricted computational model is used by Lenz [20] (see also [21, Part II]). Chan and Chen [10] present algorithms that have read-only random access to the input and are allowed only sublinear (but super-constant) space. They give a randomized linear-time algorithm for finding the convex hull of n sorted points in the plane with $O(n^\delta)$ space, for any $\delta > 0$. They also show how to perform linear programming in constant dimension, using $O(n)$ expected time and $O(\log n)$ *cells* of work space.

Throughout the paper, we assume that the input is in general position: no two points have the same x- or y-coordinates; no three points are collinear; no four points are cocircular; and all pairwise distances are distinct. For each of our algorithms, there should be a straightforward yet tedious way to remove this assumption. However, we leave it as an open problem to adapt generic methods for removing general position assumptions in the constant-work-space model.

Organization. This paper is organized as follows. After briefly describing our computational model in Section 2, we present a constant-work-space plane sweep algorithm for finding a triangulation of a planar point set and for computing the trapezoidation of a planar polygon in Section 3. In Section 4, we describe a collection of operations to compute the features by directly scanning the input. We present two example applications of such operations, one for the Delaunay triangulation and one for the Voronoi diagram of a planar point set. The operations are applied to computing the Euclidean Minimum Spanning tree. In Section 5, we give a constant-work-space algorithm for finding a shortest geodesic path between two points in a simple polygon. We conclude with some open problems.

2 Computational Model

In this section we describe our computational model. The input is stored in a read-only array, where each cell contains a data word of $O(\log n)$ bits. Although the algorithm may not permute the array elements or modify the content of an input cell, constant-time random access to the data is possible. Furthermore, we assume that any basic arithmetic operation takes constant time. Additionally, a constant-work-space algorithm can use at most some constant number of variables, each with $O(\log n)$ bits. Implicit storage consumption required by recursive calls is also considered a part of the work-space.

As a simple example of a constant-work-space algorithm, consider the problem of computing the convex hull of a planar point set. Here, an efficient such algorithm is already known. It is the popular gift-wrapping method, also known as Jarvis' march [12, 18]. If h is the number of points on the convex hull, then Jarvis' march needs $O(nh)$ time and constant work-space to output all the edges of the convex hull.

3 Constant-Work-Space Plane Sweep

Plane sweep is one of the most widespread algorithmic techniques in computational geometry. The idea is to reduce a two-dimensional problem to a sequence of one-dimensional problems by imagining a line that moves across the plane and by maintaining the intersection of that line with the structure of interest. A number of geometric problems have been solved using this paradigm [6,12,24]. Most of these solutions run in $O(n \log n)$ steps and use a balanced search tree for maintaining the one-dimensional structure as it evolves over time.

3.1 Triangulation of a Point Set

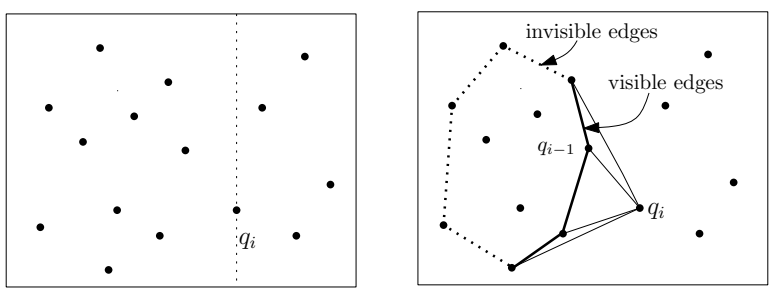
As a warm-up, we describe a constant-work-space algorithm that uses the plane sweep paradigm in order to compute a triangulation of a planar n-point set S. A straightforward cubic time algorithm adds edges incrementally: we start with a graph having n isolated vertices (points). For every pair of points, if the line segment between the two points does not properly intersect any existing edge (line segment), we add this edge to the graph. In the constant-work-space setting, however, we cannot remember the existing edges, so this simple approach will not work. Fortunately, plane sweep enables us to design a quadratic-time algorithm with only constant work-space.

The main idea is to sweep over the point set in non-decreasing order of x-coordinate, applying the gift-wrapping algorithm for convex hulls in order to add the triangles for the next point. Refer to Algorithm 1. The input consists of a list $\langle p_1, \ldots, p_n \rangle$ of points in the plane, given in no particular order. Our algorithm scans S in non-decreasing order of x-coordinate. For each point q_i in the sorted order, we compute a partial convex hull for the points $q_1, \ldots, q_{i-1}$ to the left of q_i, using the gift-wrapping method. We start the gift-wrapping from the point q_{i-1} just preceding q_i in x-order, and we extend the convex hull in both directions, upward and downward. Whenever we discover a new convex hull edge e, we determine whether the edge is visible from q_i or not, using the preceding convex hull edge (see Figure 1). If e is visible from q_i, we report the triangle spanned by q_i and e and proceed to the next hull edge. Otherwise, we stop the gift-wrapping in this direction. Once the gift-wrapping is completed

Algorithm 1: A constant-work-space algorithm for triangulating a planar point set.

Input: A set $S = \{p_1, \ldots, p_n\}$ of n points.
Output: All the triangles in a triangulation of S.
Find the three leftmost points q_1, q_2, q_3 in S.
Report the triangle $\triangle(q_1, q_2, q_3)$.
for $i := 4$ **to** n **do**
  $q_i :=$ the leftmost point in S to the right of q_{i-1}.
  $u := q_{i-1}$.
  repeat
    $e :=$ the clockwise next hull edge (u, v) incident to u.
    if *e is visible from q_i* **then**
      Report the triangle $\triangle(q_i, u, v)$ and set $u := v$.
  until *edge e is not visible from q_i*
  $u := q_{i-1}$.
  repeat
    $e :=$ the counterclockwise next hull edge (u, v) incident to u.
    if *e is visible from q_i* **then**
      Report the triangle $\triangle(q_i, u, v)$ and set $u := v$.
  until *edge e is not visible from q_i*

in both directions, we proceed to the next point in x-order, q_{i+1}.



(a) (b)

Figure 1: Triangulation of a point set via plane sweep: (a) the input and the i-th point q_i in the sorted order; (b) we compute the convex hull of the points to the left of q_i and report the triangles defined by q_i and the hull edges visible from q_i.

Theorem 3.1. *There exists an algorithm that outputs all triangles of a triangulation of S in time $O(n^2)$ and using $O(1)$ cells of work-space.*

Proof. To prove correctness, we proceed by induction on the number of points n. If $n = 3$, there is only one triangle to report. For $n \geq 4$, let q_n be the rightmost point in S. By induction, Algorithm 1 correctly outputs a triangulation for $S \setminus \{q_n\}$. This triangulation lies inside the convex hull of $S \setminus \{q_n\}$. By the time q_n is considered, in the last iteration of the **for**-loop, the algorithm outputs all possible triangles that connect q_n to the convex hull of $S \setminus \{q_n\}$, so none of these triangles can conflict with any of the previous triangles, resulting in a correct triangulation of S.

Let us now analyze the running time. Clearly, finding the initial triangle $\triangle(q_1, q_2, q_3)$ takes $O(n)$

steps. Next, we claim that if an iteration of the **for**-loop outputs k triangles, then it takes $O(kn)$ steps. First, finding q_i given q_{i-1} takes linear time, by scanning the whole input. Then each gift wrapping step takes $O(n)$ time, and for each such step, except the final ones, we output a triangle. Thus, since there are $O(n)$ triangles in total, the resulting running time is $O(n^2)$. Furthermore, inspecting the pseudocode, we see that the algorithm uses only constant work-space. □

In Section 4.1, we strengthen Theorem 3.1 by showing that the features of a *Delaunay* triangulation can be enumerated in quadratic time and with constant work-space.

3.2 Trapezoidation of a Polygon

Now let P be a simple polygon with n vertices. The input is provided as a list of vertices, given according to the counterclockwise order in which they appear along the boundary of P, ∂P. The *trapezoidal decomposition* of P is obtained by drawing two vertical rays from each vertex of P inside P, until they reach ∂P [6, Chapter 6]. This results in a collection of disjoint trapezoids that cover the interior of P. We now show how plane sweep can be used to compute these trapezoids in quadratic time with constant work-space. Refer to Algorithm 2. We scan the n vertices from left to right. At each vertex q_i, we check if there is a trapezoid to the right of q_i incident to q_i. This happens precisely if at least one of the two edges incident to q_i has an endpoint to the right of q_i. If the test is positive, we first compute two polygon edges: e_A just above q_i and e_B just below q_i. This is done by traversing all of P. Here, an edge e is *above* q_i if it intersects the upward vertical ray from q_i, or, in case that e is incident to q_i, if e has an endpoint to the right of q_i and the interior of the polygon lies below e. An edge e being *below* q_i is defined analogously. Next, we inspect all vertices of P to find the right side of the trapezoid: we start with the leftmost vertex among the two right endpoints of the edges e_A and e_B, and we update the right vertex if we find a polygon vertex inside the current trapezoid. Figure 2 illustrates how the algorithm proceeds.

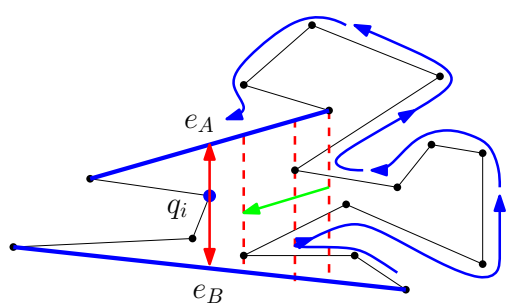


Figure 2: Computing the trapezoid to the right of the vertex q_i: we first find the two bounding edges e_A and e_B, and then we compute the right side by traversing the polygon.

Theorem 3.2. *There exists an algorithm that outputs all trapezoids of a trapezoidal decomposition of S in time $O(n^2)$ and using $O(1)$ cells of work-space.*

Proof. Correctness is immediate from the definition of a trapezoidal decomposition. For the running time, we note that each iteration of the main **for**-loop can easily be performed in linear time, since it involves two scans over all vertices of P. Again, the work-space requirement is immediate from the pseudo-code. □

Algorithm 2: A constant-work-space algorithm for the trapezoidal decomposition of a simple polygon.

```
Input: A simple polygon P = p_1 p_2 ... p_n with n vertices.
Output: All the trapezoids in a trapezoidation of P.
for i := 1 to n do
    if i = 1 then
        q_i := the leftmost vertex of P.
    else
        q_i := the leftmost vertex in P to the right of q_{i-1}.
    if there is a trapezoid to the right of q_i incident to q_i then
        e_A, e_B := ⊥.
        for j := 1 to n do
            if p_j p_{j+1} lies above q_i and (e_A = ⊥ or p_j p_{j+1} lies below e_A) then
                e_A := p_j p_{j+1}.
            if p_j p_{j+1} lies below q_i and (e_B = ⊥ or p_j p_{j+1} lies above e_B) then
                e_B := p_j p_{j+1}.
        r := leftmost of the right endpoints of e_A and e_B.
        for j := 1 to n do
            if p_j lies in the trapezoid defined by q_i, e_A, e_B, and r then r := p_j.
        Report the trapezoid defined by q_i, e_A, e_B, and r.
```

4 Constant Work-space Operations for Delaunay triangulations and Voronoi diagrams

In this section, we present constant-work-space operations for the Delaunay triangulation and the Voronoi diagrams of a planar point set. We support the usual operations of the DCEL data structure (see, for example, [6]), with constant work-space. In particular, the operations are implemented by a single scan over the input data. As an application, we present a constant-work-space algorithm for enumerating the edges of the Euclidean minimum spanning tree of a planar point set.

4.1 Operations for Delaunay Triangulations

One of the most popular structures in computational geometry is the *Delaunay triangulation* [6, 24]. For a planar n-point set S, the Delaunay triangulation of S, $\mathrm{DT}(S)$, is a triangulation of S with the *empty circle property*: for any triangle t in $\mathrm{DT}(S)$, the circumcircle of t contains no points of S in its interior. It is well known that $\mathrm{DT}(S)$ always exists and that it is uniquely defined if no four points in S lie on a common circle. If $\Theta(n)$ work-space is available, there are several algorithms to compute $\mathrm{DT}(S)$ in $O(n \log n)$ time [6, 24].

Let us now list some well-known facts about $\mathrm{DT}(S)$ that will be used [6, Chapter 9]. For two points p_i, p_j in S, we call the line segment p_ip_j a *Delaunay edge* if $\mathrm{DT}(S)$ contains p_ip_j as an edge.

Observation 4.1. *Two points $p_i, p_j \in S$ define a Delaunay edge if and only if there is a point $p_k \in S \setminus \{p_i, p_j\}$ such that the circle through p_i, p_j, p_k does not contain any other point of S in its interior.*

Observation 4.2. *Let $p_i \in S$, and let $p_j \in S \setminus \{p_i\}$ be the point closest to p_i. Then p_ip_j is a Delaunay edge.*

Observation 4.3. *Every edge of the convex hull of S is a Delaunay edge.*

We support the following operations on the Delaunay triangulation:

- *FirstDelaunayEdgeIncidentTo*(p): returns the first Delaunay edge incident to a point $p \in S$;
- *ClockwiseNextDelaunayEdge*(pq): returns the clockwise next Delaunay edge incident to p following the Delaunay edge pq;
- *CounterclockwiseNextDelaunayEdge*(pq): returns the counterclockwise next Delaunay edge incident to p following the Delaunay edge pq;
- *AssociatedDelaunayTriangle*(pq) : returns the Delaunay triangle associated with a Delaunay edge pq (i.e., the Delaunay triangle incident to pq in clockwise direction around p); and
- *NextDelaunayEdgeOnBoundary*(pq): returns the next Delaunay edge on the same facet (Delaunay triangle in this case) of pq.

FirstDelaunayEdgeIncidentTo(p) is implemented by finding the point $q \in S$ that is closest to p. The edge pq is guaranteed to be a Delaunay edge by Observation 4.2.

ClockwiseNextDelaunayEdge(pq) is implemented as follows. We distinguish two cases depending on whether there exists a point $r \in S$ to the right of the directed line ℓ_{pq} from p to q. If this is the case, then for each such point r, the triple (p, q, r) constitutes a clockwise turn. We iterate over all these points r, in arbitrary order. In each step, we keep a *tentative triangle* $\triangle(p, q, r)$, initialized with the first point r to the right of ℓ_{pq}. If the next point r' to the right of ℓ_{pq} lies inside the circumcircle of $\triangle(p, q, r)$, we replace r by r' and continue with the new tentative triangle $\triangle(p, q, r')$. If not, we keep r and $\triangle(p, q, r)$. The properties of the Delaunay triangulation ensure that the final tentative triangle $\triangle(p, q, r)$ is a Delaunay triangle (when r' is the endpoint of the clockwise next Delaunay edge, the circumcircle-test must be positive, because $\triangle(p, q, r')$ is a Delaunay triangle for the set $\{p, q, r, r'\}$; once the tentative triangle is a Delauany triangle, all further circumcircle-tests will be negative). Thus, we can find the next edge pr by a single scan over S.[1]

On the other hand, if S contains no point to the right of pq (an edge of the convex hull of S in this case), we have to find the next convex hull edge incident to p. This is also rather easy. We only need to find the point r of S that maximizes the angle $\angle qpr$. Then, the next edge is pr. See Figure 3.

Given an edge pq, we execute the two algorithms above simultaneously while checking whether there is a point to the right of the line $\overline{pq}$. When the scan is over, we know whether the edge is on the convex hull or not and hence which output we need to take. Thus, a single scan suffices.

CounterclockwiseNextDelaunayEdge(pq) is just symmetric.

[1]The algorithm in the published version of this article is slightly different and works with the signed area of $\triangle(p, q, r)$. Unfortunately, this does not always give the correct result. We thank Mika Delor for pointing this out to us.

The fourth operation, *AssociatedDelaunayTriangle*(pq) is easy since it suffices to execute *ClockwiseNextDelaunayEdge*(pq). Then, we return the triangle $\triangle(p, q, r)$.

Finally, *NextDelaunayEdgeOnBoundary*(pq) is also implemented using *ClockwiseNextDelaunayEdge*(pq). Once we have the Delaunay triangle $\triangle(p, q, r)$, we return the edge qr.

Lemma 4.4. *There are constant-work-space algorithms implementing the five operations above in $O(n)$ steps. The algorithms each perform only a single scan over the points in S.*

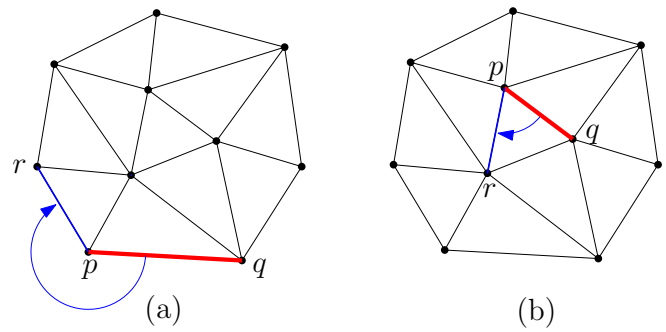


Figure 3: The two cases for clockwise next Delaunay edges. (a) pq is a hull edge; (b) pq is an internal edge.

Now the algorithm for computing the Delaunay triangulation $\mathrm{DT}(S)$ of a point set S is as follows:

Algorithm 3: A constant-work-space algorithm for computing the Delaunay triangulation of a planar point set.

Input: A set S of n points, $\{p_1, \ldots, p_n\}$.
Output: All triangles in the Delaunay triangulation of S.
for *each point* $p_i \in S$ **do**
  $p_ip_j =$ *FirstDelaunayEdgeIncidentTo*(p_i).
  // Find the point $p_j \in S$ that is nearest to p_i.
  $j_0 := j$.
  repeat
    $p_ip_k :=$ *ClockwiseNextDelaunayEdge*(p_ip_j).
    if $i < j$ *and* $i < k$ **then** Report the triangle $\triangle(p_i, p_j, p_k)$.

    $j := k$.
  until $j = j_0$.

By reporting triangles only if $i < j, k$, we avoid duplicate outputs. Figure 4 illustrates the basic operation in our algorithm. We have thus proved the following theorem:

Theorem 4.5. *Let S be a planar n-point set. There is an algorithm that reports every triangle in $\mathrm{DT}(S)$ exactly once in $O(n^2)$ time using constant work-space.*

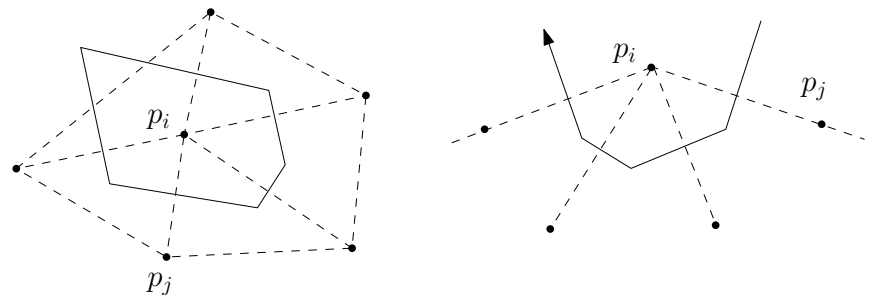


Figure 4: Going around an internal point p_i starting from its closest point p_j (left) and around an extreme point (right). Delaunay and Voronoi edges are drawn as dotted and solid lines, respectively.

4.2 Operations for Voronoi Diagrams

Given a planar point set S, the *Voronoi diagram* of S is a partition of the plane into *Voronoi regions*, such that each Voronoi region contains all the points in the plane that have the same nearest neighbor in S. The Voronoi regions are convex polygonal regions, bounded by Voronoi edges and Voronoi vertices. Since Delaunay triangulations and Voronoi diagrams are dual to each other [6, 24], it is rather easy to adapt the algorithms from Section 4.1 for Voronoi diagrams.

Lemma 4.6. *There are constant-work-space algorithms implementing the following four operations in linear time:*

- *FirstVoronoiVertexAssociatedWith*(p)*: returns the first Voronoi vertex on the boundary of the Voronoi region associated with the point* $p \in S$*;*
- *ClockwiseNextVoronoiEdge*$(e = (u, v), u = (p, q, r), v = (p, q, s))$*: returns the next Voronoi edge incident to* u*, following* e *in the clockwise direction;*
- *CounterClockwiseNextVoronoiEdge*$(e = (u, v), u = (p, q, r), v = (p, q, s))$*: symmetric to the above;*
- *NextVoronoiEdgeOnBoudary*$(e = (u, v), u = (p, q, r), v = (p, q, s))$*: returns the Voronoi edge following* e *along the boundary of its Voronoi region.*

If we represent a Voronoi vertex by the triple that defines the corresponding empty circle and a Voronoi edge by a pair of triples for its two endpoints, it is easy to see that the operations in Lemma 4.6 can be implemented immediately by using operations on the Delaunay triangulation. As can be seen from Section 4.1, every operation needs just a single scan over the input S.

4.3 Euclidean Minimum Spanning Trees

As an application of Algorithm 3, consider the problem of constructing the Euclidean minimum spanning tree (EMST) of a planar point set S. It is well-known that the EMST is a subgraph of the Delaunay triangulation [6]. Furthermore, a Delaunay edge uv is not contained in the EMST if and only if $\mathrm{DT}(S)$ contains a path between u and v consisting of Delaunay edges of length less than $d(u, v)$. This follows from the so-called *bottleneck shortest path property* of minimum spanning trees, which says that the minimum spanning tree of a graph G connects any two vertices u, v in G with a path that

minimizes the maximum edge length for any path between u and v (since otherwise we could obtain a cheaper spanning tree by exchanging an edge) [14].

Thus, consider the following situation: we are given an edge pq of $\mathrm{DT}(S)$ and we want to determine whether pq appears in the Euclidean MST. By the above observation, for this it suffices to determine whether p and q are connected in the subgraph G' of $\mathrm{DT}(S)$ that contains all Delaunay edges of length less than $d(p,q)$. To solve this problem, we could use Reingold's constant-work-space algorithm for st-connectivity in undirected graphs [25]. However, this algorithm is quite involved and has a comparatively large running time, so we present an alternative quadratic-time solution that exploits the planar structure of the Delaunay triangulation. Our algorithm is based on the following observation: let G' be the subgraph defined above, and let f be the face of G' containing pq. If p and q are connected in G', then there exists a path from p to q along the boundary of f. Thus, we can check the existence of such a path by using the function *ClockwiseNextDelaunayEdge*(uv) from Section 4.1 to follow f's boundary.

This is implemented in the function *CheckMSTEdge*. The function walks along the boundary of f, using two edges (u_A, v_A) and (u_B, v_B). Initially, both edges are set to (p,q). In a loop, we advance the edges along the boundary, using the function *Advance*. This function calls *ClockwiseNextDelaunayEdge* to find the next edge that is shorter than pq, and then follows this edge. The walk along the boundary of f continues until we reach the vertex q, in which case pq cannot be an MST edge, or until we detect a cycle, meaning that p and q are not connected in G'. The cycle is detected by using the standard technique called "baby-step, giant-step": the edge (u_B, v_B) is advanced twice as fast as the edge (u_A, v_A). If there is a cycle, (u_B, v_B) must overtake (u_A, v_A) after some time. If not, u_B reaches q first.

Function CheckMSTEdge(p,q) for checking whether the Delaunay edge (p,q) appears in the EMST.

```
Function CheckMSTEdge(p, q)
(u_A, v_A), (u_B, v_B) := (p, q).
repeat
    (u_B, v_B) := Advance(u_B, v_B).
    if u_B = q then return FALSE.

    if (u_A, v_A) = (u_B, v_B) then return TRUE.
    (u_B, v_B) := Advance(u_B, v_B).
    if u_B = q then return FALSE.

    if (u_A, v_A) = (u_B, v_B) then return TRUE.
    (u_A, v_A) := Advance(u_A, v_A).
until FALSE
Function Advance(u, v)
u, w := ClockwiseNextDelaunayEdge(uv).
if d(u, w) < d(p, q) then
    // Proceed to w.
    return (w, u).
else
    // Skip edge (u, w) and look for next clockwise edge.
    return (u, w).
```

Lemma 4.7. *The function CheckMSTEdge(p, q) determines whether the Delaunay edge pq is an edge of the Euclidean minimum spanning tree for S in time $O(n^2)$ using constant work-space.*

Proof. There are two cases to consider. If there is a path in the subgraph of the Delaunay triangulation defined by those edges shorter than pq which interconnects p and q, the path forms a face together with the edge pq. The first edge incident to p on the boundary is given by the clockwise next Delaunay edge of pq. Then, following the boundary we must reach q, which is detected by the algorithm.

On the other hand, if there is no such path between p and q, then at some point such a path does not extend, which causes a cycle of edges.

Furthermore, *CheckMSTEdge*(p, q) visits a subset of edges (u, v) of the Delaunay triangulation, where each edge is visited constantly often. Each visit requires a call to *ClockwiseNextDelaunayEdge*(uv), which takes $O(n)$ time, by Lemma 4.4. Thus, the total running time is $O(n^2)$. The space requirement is immediate. □

With the function *CheckMSTEdge* at hand, we can output all edges of the Euclidean MST for S through a straightforward adaptation of Algorithm 3.

Algorithm 4: A constant-work-space algorithm for computing the Euclidean minimum spanning tree of a planar point set.

Input: A set S of n points, $\{p_1, \ldots, p_n\}$.
Output: All edges in the Euclidean MST of S.
for *each point* $p_i \in S$ **do**
  Find a point $p_j \in S \setminus \{p_i\}$ that is nearest to p_i.
  $j_0 := j$.
  repeat
    if $i < j$ **and** *CheckMSTEdge*(p_i, p_j) **then** Report p_ip_j.

    $p_k :=$ ClockwiseNextDelaunayEdge(p_ip_j).
    $j := k$.
  until $j = j_0$.

Theorem 4.8. *Given a set S of n points in the plane, there exists an algorithm that outputs all edges of the Euclidean minimum spanning tree for S in $O(n^3)$ time using only constant work-space.*

Proof. Correctness follows from Observation 4.2 and Lemmas 4.4 and 4.7. For the running time, note that it takes total time $O(n^2)$ to find the nearest neighbor for every point in S, and since there are $O(n)$ edges in DT(S), the total time for all invocations of *ClockwiseNextDelaunayEdge* and *CheckMSTEdge* is $O(n^2 + n^3) = O(n^3)$, by Lemmas 4.4 and 4.7. The space requirement is immediate. □

Figure 5 illustrates the steps of our algorithm The edge pq in the top of the figure is not included in the EMST, since p and q are connected by a path of Delaunay edges shorter than pq. Figure 5(c) shows how such a path is found by walking along the face boundary. On the other hand, the bottom part of the figure shows the other case in which the walk results in a cycle.

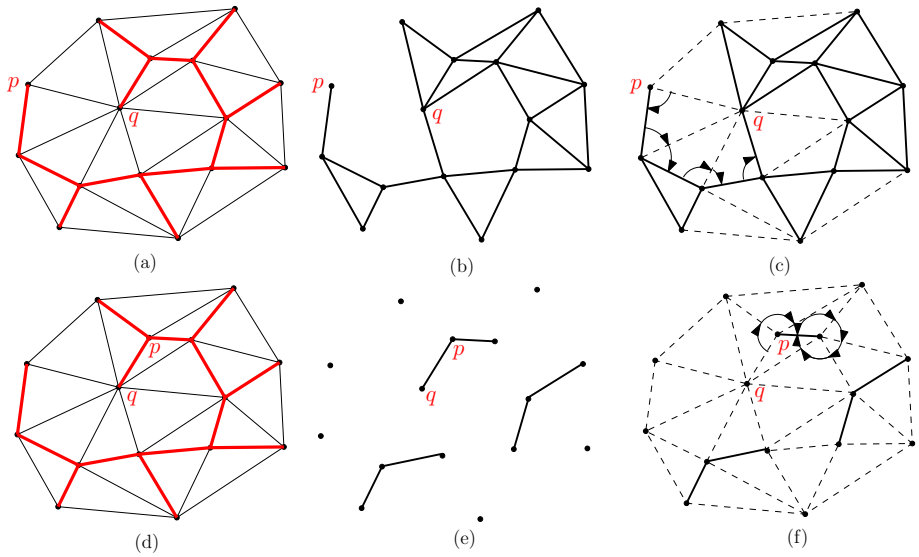


Figure 5: Checking whether a Delaunay edge belongs to the minimum spanning tree. (a) a Delaunay triangulation and a minimum spanning tree (bold lines) with two specified points p and q. (b) The Delaunay edges shorter than the edge pq. (c) We traverse the subgraph until we find a path between p and q. (d) A different point pair (p, q). (e) The Delaunay edges shorter than pq. (f) We traverse the subgraph until we find a loop.

5 Computing a Shortest Path in a Simple Polygon

As a final problem, we consider the *geodesic shortest path problem* for simple polygons. The general problem of finding a shortest path between two given vertices in a weighted graph is a classic of algorithm design, and countless solutions exist, such as the well-known algorithms by Dijkstra and Bellman-Ford [12]. In the constant-work-space model, however, no algorithm for the general shortest path problem is known, and it is unlikely that such an algorithm does exist, since the problem turns out to be NL-complete [27].[2]

However, in this section we will see that a special case of the shortest path problem can be solved in quadratic time with constant work-space: the geodesic shortest path problem within a simple polygon. Here, we are given a simple polygon P with n vertices and two points s and t in the interior of P, and we are looking for the shortest path from s to t that lies within P. This problem can be solved in linear time with $\Theta(n)$ work-space [19].

We now describe our algorithm. We assume that P is given as a counterclockwise sequence of vertices in a read-only array and that the two points s and t lie inside P. Due to our general position assumption, no three vertices lie on a line. The shortest path from s to t can be represented as a sequence $\langle s = v_0, v_1, \ldots, v_m = t\rangle$, where v_1, …, v_{m-1} are vertices of P, and the algorithm will output these vertices in order.

Throughout the algorithm, we maintain a polygon vertex p as our current starting point, as

[2] NL is the class of all decision problems that can be solved by a *non-deterministic* constant-work-space algorithm. A problem in NL is *NL-complete* if all problems in NL can be reduced to it by a deterministic constant-work-space reduction. It is widely conjectured that NL $\neq$ L, and that NL-complete problems cannot be solved by a deterministic constant-work-space algorithm [1].

well as two points q_1 and q_2 on the boundary of P, ∂P, such that the line segments pq_1 and pq_2 lie inside P (possibly touching the boundary) and such that pq_2 is counterclockwise from pq_1. Note that pq_1 and pq_2 could be edges. This defines a region $P' \subseteq P$ that is cut off by pq_1 and pq_2, as shown in Figure 6. The algorithm proceeds by advancing the triple (p, q_1, q_2) while maintaining the following

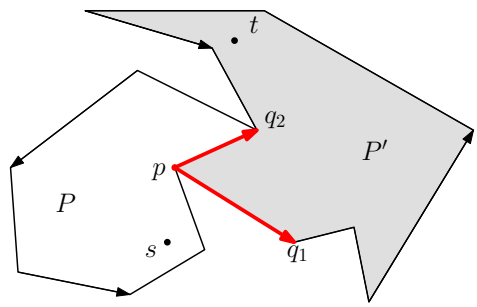


Figure 6: A current starting point p and two supporting line segments pq_1 and pq_2. Together, they define a subpolygon P' which contains the target t.

invariant:

Invariant 5.1. *(i) The geodesic shortest path from s to t passes through p.*
(ii) t lies in the subpolygon P'.

The triple (p, q_1, q_2) is advanced by a function *MakeStep*(p, q_1, q_2), such that in (almost) every step the subpolygon P' becomes smaller.

MakeStep distinguishes three cases.

Case 1: q_1 is a concave vertex of P', that is, $p, q_1, \mathrm{succ}(q_1)$ is a clockwise turn, where $\mathrm{succ}(q)$ is the successive vertex of q on ∂P. Refer to Figure 7. In this case we extend the ray pq_1 until it hits $\partial P'$

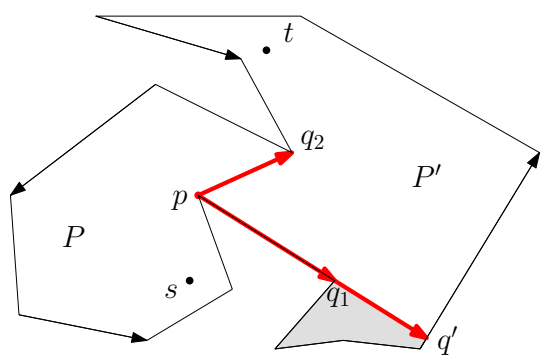


Figure 7: Case 1: the vertex q_1 is concave.

at q'. The segment q_1q' splits P' into two parts. We check which one contains t and update (p, q_1, q_2) accordingly. If the path needs to turn at p, we also output p. Note that we can check in constant time which subpolygon contains t, assuming we precomputed the index of the edge e just above the target t. Indeed, using this edge index, we look up the coordinates of the endpoints of e in constant time. Then we determine whether the vertical line segment from t to e intersects q_1q'. If yes, we immediately know which subpolygon contains t. Otherwise, we can determine this polygon using vertex indices, because t must be in the same subpolygon as e.

Function MakeStep(p, q_1, q_2) for advancing the triple (p, q_1, q_2) while maintaining Invariant 5.1.

```
Function MakeStep(p, q_1, q_2)
begin
    if q_1 is a concave vertex of P' then
        Traverse P to find the point q' where the ray pq_1 first meets ∂P.
        if t lies in the subpolygon from q' to q_1 then
            Output p.
            return (q_1, succ(q_1), q').
        else
            return (p, q', q_2).
    else if q_2 is a concave vertex of P' then
        Traverse P to find the point q' where the ray pq_2 first meets ∂P.
        if t lies in the subpolygon from q_2 to q' then
            Output p.
            return (q_2, q', pred(q_2)).
        else
            return (p, q_1, q').
    else
        if the ray p succ(q_1) lies in the wedge q_1pq_2 then
            Traverse P to find the first intersection q' of the ray p succ(q_1) with ∂P.
            if t lies in the subpolygon from q' to q_1 then
                return (p, q_1, q').
            else
                return (p, q', q_2).
        else
            Traverse P to find the first intersection q' of the ray p pred(q_2) with ∂P.
            if t lies in the subpolygon from q_2 to q' then
                return (p, q', q_2).
            else
                return (p, q_1, q').
```

Case 2: q_2 is a concave vertex of P', that is, $p, q_2, \text{pred}(q_2)$ constitutes a counterclockwise turn, where $\text{pred}(q)$ is the predecessor of q on ∂P. This case is handled symmetrically to Case 1.

Case 3: The polygon P' makes a convex turn at both q_1 and q_2. In particular, q_1 and/or q_2 could lie in the interior of edges of P. Refer to Figure 8. Let $\text{succ}(q_1)$ and $\text{pred}(q_2)$ be the neighboring vertices

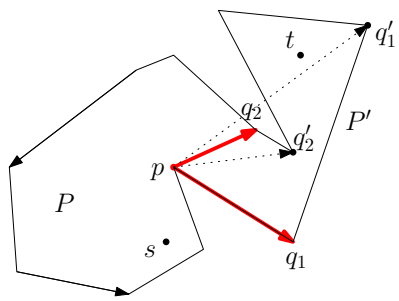


Figure 8: Case 3: both q_1 and q_2 are convex. Then, one of $(q_1, \text{succ}(q_1))$ and $(q_2, \text{pred}(q_2))$ lies between pq_1 and pq_2.

of q_1 and q_2 in P'. At least one of the rays $p\,\text{succ}(q_1)$ and $p\,\text{pred}(q_2)$ lies within the wedge defined by q_1pq_2; otherwise the edges $q_1\,\text{succ}(q_1)$ and $q_2\,\text{pred}(q_2)$ would intersect. Suppose this is $p\,\text{succ}(q_1)$. We draw the ray from p toward the point $\text{succ}(q_1)$ until it hits ∂P at some point q' (q' could be $\text{succ}(q_1)$). As in the previous two cases, q_1q' splits P' into two parts. We check which one contains t as before, and then update (p, q_1, q_2) accordingly. If the ray $p\,\text{pred}(q_2)$ lies inside the wedge q_1pq_2, we proceed symmetrically.

Lemma 5.2. *MakeStep maintains Invariant 5.1 and takes $O(n)$ time and constant work-space.*

Proof. All the steps in *MakeStep* can be performed in time $O(n)$ and constant work-space, as can be seen by inspecting the pseudo-code. It is also clear from the case analysis that Invariant 5.1 is maintained. □

With the function *MakeStep* in place, it is now easy to implement the geodesic shortest path algorithm (Algorithm 5). In order to initialize the tripe (p, q_1, q_2), Algorithm 5 needs to locate the point s within P. For this, we invoke Algorithm 2 to find the trapezoid T_0 that contains s, and initialize (p, q_1, q_2) accordingly. The vertex p is set to s, and q_1, q_2 are set depending on the structure of T_0 and which of the at most four subpolygons defined by T_0 contains t; see Figure 9. We also use Algorithm 2 to identify the segment e_t on ∂P right above t, which is later used to determine which subpolygon contains t. Now we just need to invoke *MakeStep* repeatedly until we reach a polygon vertex that can see t directly.

Theorem 5.3. *Let P be a simple polygon with n vertices, and let s and t be two points within P. Then a geodesic shortest path from s to t within P can be found in $O(n^2)$ time using only constant work-space.*

Proof. By Theorem 3.2, the initialization phase takes $O(n^2)$ time and constant work-space. Furthermore, the check at the end of the **repeat-until** loop can be performed in time $O(n)$ and constant work-space. Thus, by Lemma 5.2, Algorithm 5 needs constant work-space, and the total running time is $O(n^2 + kn)$, where k is the number of calls to *MakeStep*. To see that $k = O(n)$, we observe that

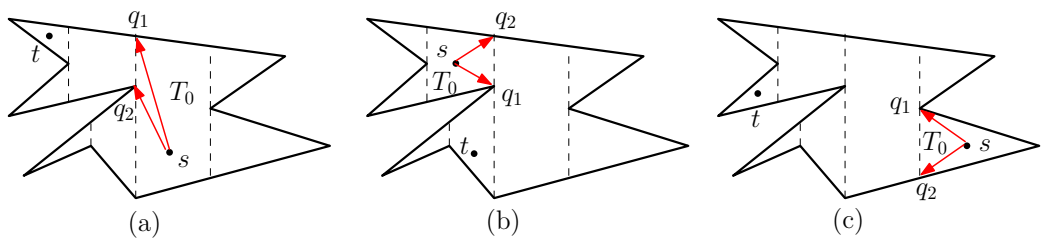


Figure 9: Three different situations for the initial 3-tuple (p, q_1, q_2) with $p = s$: the trapezoid containing s can have (a) four, (b) three, or (c) a single neighbor.

Algorithm 5: A constant-work-space algorithm for finding a geodesic shortest path within a simple polygon.

Input: A simple polygon $P = p_1p_2 \ldots p_n$ with n vertices; two points s and t inside P.
Output: A sequence $s = v_0v_1 \ldots v_{m-1}v_m = t$ of the vertices of a shortest path from s to t in the interior of P.

```
Invoke Algorithm 2 to enumerate a trapezoidal decomposition of P.
for each trapezoid T returned by Algorithm 2 do
    if s ∈ T then
        if t ∈ T then
            Output s, t.
            return
        else
            T_0 := T.
    if t ∈ T then
        e_t := upper segment of T.
Determine which subpolygon defined by T_0 contains t and initialize q_1, q_2 accordingly.
p := s.
repeat
    (p, q_1, q_2) := MakeStep(p, q_2, q_2).
until the line segment pt does not intersect ∂P.
Output t.
```

every other call to *MakeStep* decreases the size of P' by one. This can be seen by an inspection of the case analysis for *MakeStep*. Cases 1 and 2 decrease the size of P' directly. The same also holds for the second subcase of Case 3. Only in the first subcase of Case 3 (when t lies inside the subpolygon from q' to q_1, or in the subpolygon from q_2 to q') might the size of P' not decrease. However, if this happens, the second subcase of Case 3 must apply in the next iteration, and we will make progress.

The correctness of the algorithm is immediate from Lemma 5.2. □

In order to perform some preliminary experiments with our shortest path algorithm, we created a prototype implementation. The program consists of about 700 lines of C code, including comments. We used LEDA [22] for drawing polygons and shortest paths to visualize the algorithm, see Figure 10 for an example. Furthermore, we did experiments on hand-crafted input polygons on a laptop with an Intel Core2 Duo CPU with 1.20GHz and 2.93 GB of RAM. We used three input polygons of different sizes, and we ran the algorithm to find shortest paths between several pairs of points inside the polygons. The results are summarized in Table 1. As can be seen, the running time depends quite significantly on the choice of endpoints for the shortest path. Furthermore, our implementation incurs some overhead for the visualization through LEDA. However, it can be discerned that in the worst case the running time grows superlinearly in the size of the input.

file name	test1	test2	test3
polygon size	41	194	427
	9 / 0.000	13 / 0.047	13 / 0.031
	12 / 0.016	16 / 0.015	30 / 0.031
	16 / 0.016	22 / 0.052	44 / 0.062
	20 / 0.015	34 / 0.016	45 / 0.047
	25 / 0.016	37 / 0.031	52 / 0.079
		46 / 0.031	61 / 0.078
		47 / 0.047	71 / 0.125
		54 / 0.047	88 / 0.141
		57 / 0.047	92 / 0.141

Table 1: Experimental results. The largest polygon (test3) has 427 vertices. Entries in the table are pairs of number of iterations to find the shortest path (= number of iterations of the **repeat**... **until**-loop in Algorithm 5) and CPU time (seconds), for several choices of endpoints s and t.

6 Concluding Remarks

We have presented constant-work-space algorithms for several geometric problems. A number of geometric problems are still open in the constant-work-space model:

(1) Given a set of n points in the plane, find the smallest enclosing circle. We can design an $O(n^2)$-time constant-work-space algorithm using the farthest-point Voronoi diagram. Is there any subquadratic-time algorithm?

(2) Given a simple polygon and a query point q in its interior, compute the visibility polygon from q in subquadratic time.

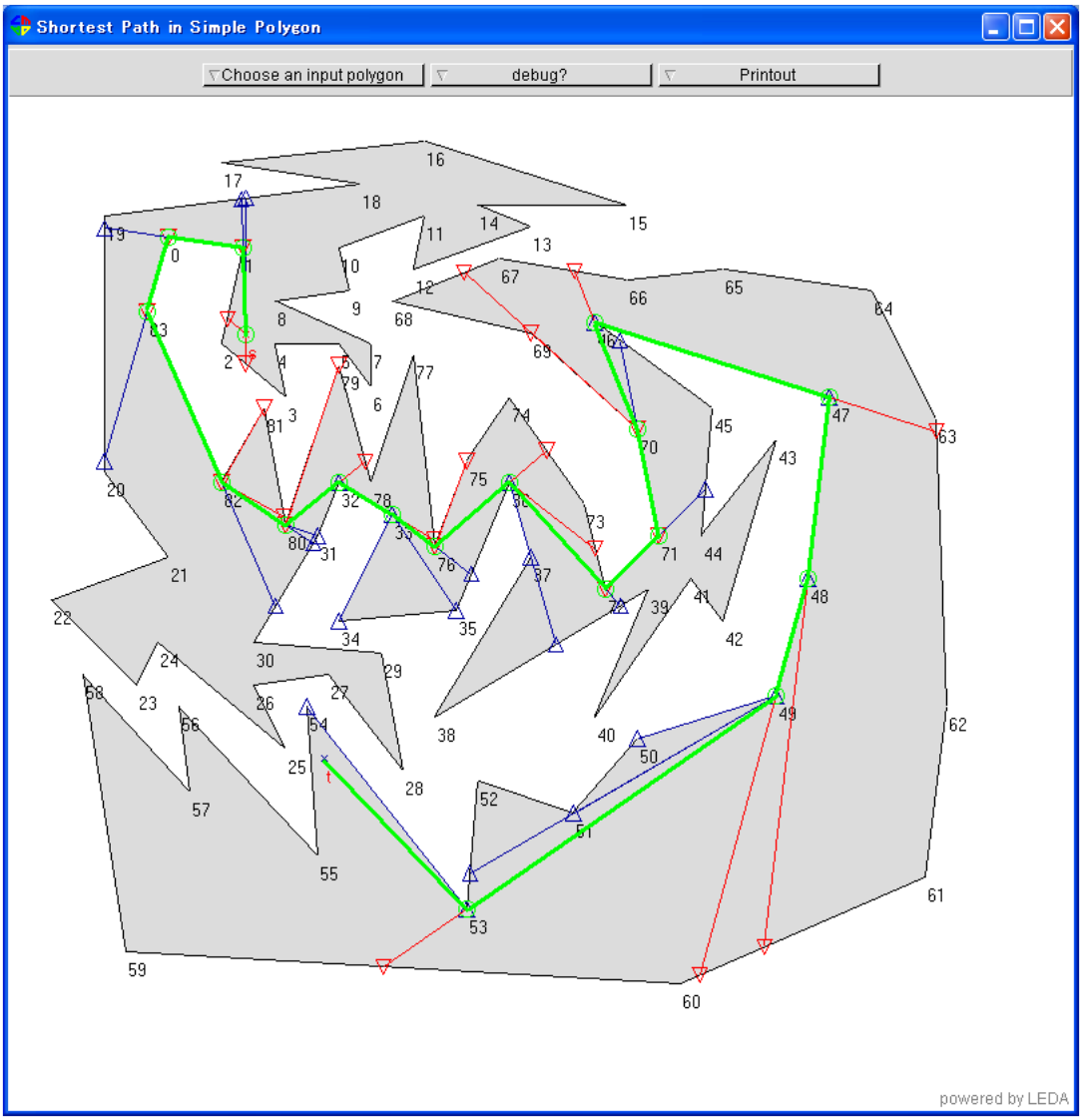


Figure 10: Shortest path in a simple polygon. The bold lines show the shortest path, and the diagonals represent the corresponding line segments pq_1 and pq_2.

(3) Given a set of points in the plane, find a largest empty circle with its center lying in the convex hull of the point set in subquadratic time. We can compute it by using our constant-work-space operations for Delaunay triangulations in quadratic time.

Another interesting direction is to investigate time-space trade-offs: how much work-space is needed to find a shortest path in a simple polygon in linear time?

So far, there are no powerful techniques for proving lower bounds with constant work-space. For the problem of approximating the median with (small) constant storage, Lenz [20] and [21, Part II] gave lower bounds in a more restricted data access model. More generally, an $\Omega(n \log n)$ lower bound is known for the element uniqueness problem in a standard computational model. Is there any higher lower bound? As far as the authors know, no subquadratic-time algorithm for the element uniqueness is known in the constant-work-space model.

Our algorithms extend to Delaunay triangulations in three dimensions, allowing to report all Delaunay edges, triangles, or tetrahedra, as well as all Voronoi vertices, edges, or faces, in polynomial time. The Euclidean minimum spanning tree can be constructed in 3-space if we use the powerful technique of Reingold [25], but it looks hard to extend Algorithm 4 to 3-space.

Acknowledgments

The work was initiated at a Dagstuhl Seminar 09111 on Computational Geometry in March, 2009. The work of T.A. was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Priority Areas and Scientific Research (B). W.M. was supported by a Wallace Memorial Fellowship in Engineering, and in part by NSF grant CCF-0634958 and NSF CCF 08327. We would like to thank the anonymous reviewers for their thorough reading of the manuscript and for numerous helpful suggestions that improved the quality of the paper.

References

[1] S. Arora and B. Barak. *Computational complexity. A modern approach.* Cambridge University Press, Cambridge, 2009.

[2] T. Asano. Constant-work-space algorithms: how fast can we solve problems without using any extra array? In *Proc. 19th Annu. Internat. Sympos. Algorithms Comput. (ISAAC), invited talk*, volume 5369 of *Lecture Notes in Computer Science*, page 1. Springer-Verlag, 2008.

[3] T. Asano. Constant-work-space algorithms for image processing. In F. Nielsen, editor, *Emerging Trends in Visual Computing (ETVC 2008)*, volume 5416 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2009.

[4] T. Asano, W. Mulzer, and Y. Wang. Constant-work-space algorithms for shortest paths in trees and simple polygons. *J. Graph Algorithms Appl.*, page to appear, 2011.

[5] D. Avis and K. Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete Comput. Geom.*, 8(3):295–313, 1992.

[6] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, third edition, 2008.

[7] M. de Berg, M. J. van Kreveld, R. van Oostrum, and M. H. Overmars. Simple traversal of a subdivision without extra storage. *International Journal of Geographical Information Science*, 11(4):359–373, 1997.

[8] P. Bose and P. Morin. An improved algorithm for subdivision traversal without extra storage. *Internat. J. Comput. Geom. Appl.*, 12(4):297–308, 2002.

[9] P. Bose and P. Morin. Online routing in triangulations. *SIAM J. Comput.*, 33(4):937–951, 2004.

[10] T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. *Discrete Comput. Geom.*, 37(1):79–102, 2007.

[11] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6(5):485–524, 1991.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, third edition, 2009.

[13] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.

[14] D. Eppstein. Spanning trees and spanners. In *Handbook of computational geometry*, pages 425–461. North-Holland, Amsterdam, 2000.

[15] C. M. Gold, T. D. Charters, and J. Ramsden. Automated contour mapping using triangular element data structures and an interpolant over each irregular triangular domain. In *Proc. 4thComput. Graph.*, pages 170–175, 1977.

[16] C. M. Gold and S. Cormack. Spatially ordered networks and topographic reconstructions. *Int. J. Geographical Information Systems*, 1(2):137–148, 1987.

[17] C. M. Gold and U. Maydell. Triangulation and spatial ordering in computer cartography. In *Proc. Canad. Cartographic Association Annual Meeting*, pages 69–81, 1981.

[18] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Inform. Process. Lett.*, 2(1):18–21, 1973.

[19] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.

[20] T. Lenz. Deterministic splitter finding in a stream with constant storage and guarantees. In *Proc. 17th Annu. Internat. Sympos. Algorithms Comput. (ISAAC)*, volume 4288 of *Lecture Notes in Computer Science*, pages 26–35, Kolkata, India, 2006. Springer-Verlag.

[21] T. Lenz. *Simple reconstruction of non-simple curves and approximating the median in streams with constant storage*. PhD thesis, Freie Universität Berlin, 2008.

[22] K. Mehlhorn and S. Näher. *LEDA. A platform for combinatorial and geometric computing*. Cambridge University Press, Cambridge, 1999.

[23] J. I. Munro and V. Raman. Selection from read-only memory and sorting with minimum data movement. *Theoret. Comput. Sci.*, 165(2):311–323, 1996.

[24] F. P. Preparata and M. I. Shamos. *Computational geometry. An introduction.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1985.

[25] O. Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):Art. #17, 24 pp., 2008.

[26] G. Rote. Degenerate convex hulls in high dimensions without extra storage. In *Proc. 8th Annu. ACM Sympos. Comput. Geom. (SoCG)*, pages 26–32, 1992.

[27] T. Tantau. Logspace optimization problems and their approximability properties. *Theoret. Comput. Sci.*, 41(2):327–350, 2007.