Constant-Working-Space Algorithms for Geometric Problems

Tetsuo Asano* Günter Rote†

Abstract

In this paper we first present an algorithm for constructing the Delaunay triangulation of n given points in the plane. It runs in $O(n^2)$ time using only constant working space (more precisely, $O(\log n)$ bits in total) with input data on a read-only array. Then, we apply it to construct a Voronoi diagram in $O(n^2)$ time and Euclidean minimum spanning tree in $O(n^3)$ time in the same model.

1 Introduction

Recent progress in computer systems has provided programmers with unlimited amount of working storage for their programs. Nowadays there are plenty of space-inefficient programs which use too much storage and become too slow if sufficiently large memory is not available. We believe that there is high demand for space-efficient algorithms.

In this paper we first present an algorithm for constructing a Delaunay triangulation for a set of points in the plane. A triangulation of a point set is a maximal planar graph in which vertices are input points and edges are straight line segments between points. The Delaunay triangulation is such a triangulation that for each triangle its circumcircle contains no other point of the set in its interior. An $O(n \log n)$ algorithm for constructing Delaunay triangulation is known [4]. This paper presents another algorithm which takes $O(n^2)$ time but requires only constant working space, that is, $O(\log n)$ bits in total. Furthermore, input data are kept in a read-only array. Outputs are not stored anywhere. Whenever we obtain an triangulation edge, we just output it as a line segment. Or we could assume a write-only array for outputs.

We can apply this algorithm to derive other space-efficient algorithms. One direct application is an algorithm for constructing the Voronoi diagram of a point set, which is a dual of Delaunay triangulation. Another application is an algorithm for constructing a minimum spanning tree. It runs in $O(n^3)$ time in the same setting.

Our algorithms take more time, but when considering the product of time and space requirements, our algorithms are an improvement. Existing algorithms for the Delaunay triangulation and the Voronoi diagram run in $O(n \log n)$ time but they need $O(n)$ working space, and hence the time and space product is $O(n^2 \log n)$. For our algorithms, the product is $O(n^2) \times O(1) = O(n^2)$. Another advantage is in their simpleness. Simpleness is also important for educational purposes.

Constant-working-space algorithms have been studied in complexity theory under a different name, "log-space" algorithms. The authors prefer the current name since it is more intuitive. One of the most important results among a number of results in log-space algorithms is a selection algorithm by Munro and Raman [7] which runs in $O(n^{1+\epsilon})$ time using working space $O(1/\epsilon)$ for any small constant $\epsilon > 0$. The result on st-connectivity on graphs by Reingold [8] is also another breakthrough in this area. See also [1, 2] for applications to image processing.

Throughout the paper, we assume for simplicity that the input is in general position: no two points have the same x or y-coordinates; no three points are collinear and no four points are cocircular; and no two point pairs have the same distance. Standard techniques like perturbation can be used to treat degenerate cases.

2 Computational Model

In this section we describe our computational model. Input data are stored in a read-only array. In our case input data consist of n points in the plane. Although it is not allowed to permute the array elements or modify the content of any element, constant-time random access to data points is possible. Further, it is assumed that any basic arithmetic operation is done in constant time.

A constant-working-space algorithm can use at most some constant number of variables, each with $O(\log n)$ bits, in addition to the read-only array for input data. Implicit storage consumption required by recursive calls is also considered as a part of working space. The same computational model is also used in [3] with a number of interesting algorithms including one for constructing the convex hull.

3 Computing a Triangulation

The straightforward approach for constructing an arbitrary triangulation adds edges incrementally: we start

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

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

with a graph having n isolated vertices (points). For every pair of points we determine whether the line segment between the two points does not properly intersect any existing edge (line segment), and if it does not then we add the edge to the graph.

In this paper we are interested in algorithms using only constant working space with input points on a read-only array. Since we have no space for marking existing edges, it is impossible to implement the naive algorithm described above in the constant working space environment. Fortunately, we can design a quadratic-time algorithm using constant working space.

Quadratic-Time Algorithm for Triangulation

There are a number of different algorithms for computing the convex hull of a point set in the plane. An algorithm known as a gift-wrapping or Jarvis' march successively finds convex hull edges in a way of wrapping a gift. For a set of n points, it takes $O(n)$ time to discover the next convex hull edge. The total running time is output-sensitive in the sense that it runs in $O(nh)$ time when the number of vertices of the convex hull is h. An advantage is that it is originally a constant working space algorithm. Using this algorithm, we can design a constant working space algorithm for constructing a triangulation.

Our algorithm scans points from left to right. Since we cannot store sorted results, we find those points one by one by finding the next point with larger x-coordinate. Then, at each point p_i we compute the convex hull for a set of point lying to its left: We discover successive hull edges by Jarvis' march, starting from the rightmost point p_{i-1} in two directions. Whenever we find a new convex hull edge, we determine whether the edge is visible from p_i or not, using its preceding convex hull edge. If it is visible, then we draw an edge from p_i to the point. The time for drawing k new edges is $O(kn)$. Therefore, the time we need for the triangulation is bounded by the number of triangulation edges times the number of points, which is $O(n^2)$. We also need time for scanning the points in order. The total time we need is thus $O(n^2)$.

4 Quadratic-Time Algorithm for Delaunay Triangulation

Here are some well-known basic observations.

Observation 1 *Given a set S of points in the plane, a line segment connecting two points p_i and p_j in S is a Delaunay edge if and only if there exists some point $p_k \in S$ such that the circle defined by them does not contain any other point of S.*

Observation 2 *If p_j is closest to p_i, then (p_i, p_j) is a Delaunay edge.*

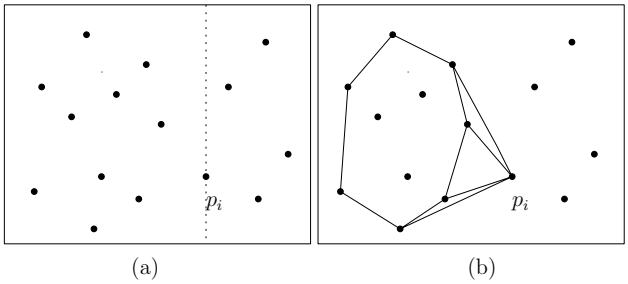

Figure 1: Triangulation of a point set using a plane sweep. (a) input points and i-th point in the sorted order, (b) computing the convex hull of points lying to the left of p_i and then construct edges to the hull points.

Observation 3 *Every edge of the convex hull of S is also Delaunay edge.*

These three observations lead to the following algorithm for constructing Delaunay triangulation. The algorithm scans each point p_i of S and constructs all the Delaunay edges incident to p_i. By Observation 2 we can start from a point p_j nearest to p_i. Then, we find the next Delaunay edge incident to p_i in clockwise order, which is found by the following procedure using Observations 1 and 3.

Algorithm 1 for finding the next edge from (u, v) in clockwise order in the Delaunay triangulation (graph) defined by a set S of points. See Figure 2

```
function ClockwiseNext(u, v)
    first := TRUE;
    for each point w ∈ S\{u, v} do
        if (u, v, w) is clockwise then
            if first or w lies in the circumcircle of u, v, w0
            then
                w0 := w;
                first := FALSE;
    if first then
        // (u, v) is a hull edge;
        // find the adjacent hull edge (u, w0):
        for each point w ∈ S\{u, v} do
            if first or (u, w0, w) is counter-clockwise then
                w0 := w;
                first := FALSE;
    return (u, w0);
```

Lemma 4 *The procedure ClockwiseNext(u, v) correctly finds the next edge incident to u in clockwise order in the Delaunay triangulation (graph) defined by a set S of points.*

Figure 3 illustrates a basic operation in our algorithm, which is successive application of the procedure *ClockwiseNext*() (Algorithm 1).

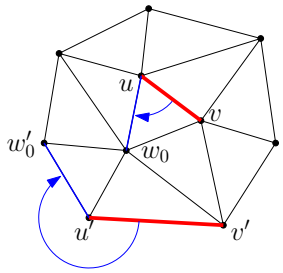

Figure 2: Clockwise next Delaunay edges, one for internal edge and the other convex hull edge.

An important observation here is that the algorithm calls the procedure *ClockwiseNext*() at most $O(n)$ times. It is because the total number of calls is bounded by the total number of Delaunay edges, which is $O(n)$. Since the procedure is done in $O(n)$ time, the total time complexity is bounded by $O(n^2)$.

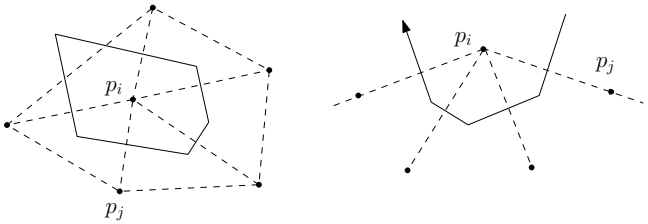

Figure 3: Go around an internal point p_i starting from its closest point p_j (left) and around an extreme point (right). Delaunay edges and Voronoi edges are drawn by dotted and solid line, respectively.

Now, the whole algorithm looks like as follows:

Algorithm 2: Constructing a Delaunay triangulation in quadratic time.

Quadratic-Time Algorithm for Constructing Delaunay Triangulation
Input: A set S of n points, $\{p_1, \ldots, p_n\}$.
Output: Delaunay edges.
for *each point* $p_i \in S$ **do**
  Find a point $p_j \in S$ that is nearest to p_i.
  Report the edge (p_i, p_j) if $i < j$.
  $p_k = p_j$.
  repeat
    p_k =ClockwiseNext(p_i, p_k).
    if $i < k$ **then** Report the edge (p_i, p_k).
  until $p_k = p_j$

By reporting edges only if $i < k$ we ensure that each edge is reported only once: when it is visited from the endpoint with the smaller number.

Theorem 5 *Given a set of n points in the plane, Algorithm 1 reports every Delaunay edge exactly once in $O(n^2)$ time using constant working space.*

5 Voronoi Diagrams

It is rather easy to extend our constant-working-space algorithm to one for constructing Vodonoi diagram of a point set. Given a set S of points in the plane, for each point u we can go around the point u while visiting all Delaunay edges in clockwise order, and hence visiting Delaunay triangles incident to u in clockwise order. Every finite Voronoi edge is defined by two adjacent Delaunay triangles. Thus, those edges can be reported by slightly modifying Algorithm 2. It is also easy to adapt Algorithm 2 to report infinite Voronoi edges, which are dual to convex-hull edges, as well.

6 Minimum Spanning Tree

As an application of the algorithm for constructing the Delaunay triangulation, consider a problem of constructing the Euclidean minimum spanning tree of a given point set. It is a well-known property that it is a subgraph of the Delaunay triangulation. Another important property of the minimum spanning tree is that a Delaunay edge $e = (u, v)$ is not contained in the minimum spanning tree if and only if the Delaunay graph contains a path between u and v consisting of Delaunay edges of lengths $< d(u, v)$. As before, we enumerate all Delaunay edges in $O(n^2)$ time in total. Whenever we have a Delaunay edge $e = (u, v)$, we check whether it is also an edge in the minimum spanning tree by checking the existence of a path between u and v consisting of Delaunay edges of lengths $< d(u, v)$. Suppose $e = (u, v)$ is not an edge in the minimum spanning tree. Then, if we add the edge (u, v) to the subgraph of the Delaunay graph which consists of all Delaunay edges of lengths $< d(u, v)$, we have a cycle, which forms a face. Thus, we can check the existence of the above-described path by successively applying the two functions *ClockwiseNext*(u, v) and *CounterClockwisewNext*(u, v) to follow the boundary of a face defined by the cycle.

The algorithm *CheckMSTreeEdge*(p, q) visits a subset of edges (u, v) of the Delaunay triangulation; each visit requires a call to *ClockwiseNext*(u, v), which takes $O(n)$ time. Thus in total, the running time is $O(n^2)$ for testing one Delaunay edge (p, q). We repeat this test for each of the $O(n)$ Delaunay edges. Thus, the time for outputting all MST edges is $O(n^3)$.

Theorem 6 *Given a set of n points in the plane, we can compute its Euclidean minimum spanning tree in $O(n^3)$ time using only constant working space.*

We could use a constant-working-space algorithm by Reingold [8] which determines connectivity of two arbitrary vertices in a given graph in polynomial time. It would be much slower and more complicated. Our algorithm is faster since it can use the planar structure of the Delaunay triangulation

Algorithm 3: Checking for an MST edge

CheckMSTreeEdge(p, q);
Input: A set $S = \{p_1, \ldots, p_n\}$ of n points, a Delaunay edge (p, q).
Output: TRUE if (p, q) is an edge of the minimum spanning tree.
// walk around the boundary of the subgraph of the Delaunay triangulation consisting of edges shorter than (p, q), starting at p in the face containing the edge (p, q);
$u, v := p, q$; // u is the current vertex and (u, v) is the current edge;
repeat
  $u, w :=$ *ClockwiseNext*(u, v);
  if $\|u - w\| < \|p - q\|$ **then**
    $u, v := w, u$; // proceed to w;
  else
    $u, v := u, w$;
    // skip edge (u, w) and look for next clockwise edge;
until $(u, v) = (p, q)$ ***or*** $u = q$;
if u=q **then**
  return FALSE;
  // The walk has reached the other endpoint q; thus p and q are connected in the subgraph;
else
  return TRUE;
  // The walk has surrounded the component containing p and reached the starting edge without reaching the other endpoint q. Thus, p and q are in different components;

7 Conclusions and Future Work

In this paper we have presented constant-working space algorithms for three geometric problems. The last one for constructing the Euclidean minimum spanning tree is based on the algorithm for constructing the Delaunay triangulation of a point set. It takes $O(n^3)$ time. Obvious future works are to improve the time complexity or to prove a lower bound and to extend the result to higher dimensions. So far, there are no techniques for proving lower bounds with constant working space. For the problem of approximating the median with (small) constant storage, Lenz [5] and [6, Part II] have given lower bounds in a more restricted data access 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. It is open whether this can be used to compute the MST, other than by using the powerful technique of Reingold [8].

Acknowledgments

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).

References

[1] T. Asano, "Constant-Working-Space Algorithms: How Fast Can We Solve Problems without Using Any Extra Array?," Invited talk at ISAAC 2008, p.1, Dec. 2008.

[2] T. Asano, "Constant-Working-Space Algorithms for Image Processing," Monograph: "ETVC08: Emerging Trends and Challenges in Visual Computing," ETVC 2008: pp.268-283, edited by Frank Nielsen, 2009.

[3] T. M. Chan, E. Y. Chen, "Multi-Pass Geometric Algorithms. Discrete & Computational Geometry 37(1), pp.79-102, 2007.

[4] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, "Computational Geometry: Algorithms and Applications," Springer-Verlag, 2008.

[5] T. Lenz, Deterministic splitter finding in a stream with constant storage and guarantees. 17th International Symposium on Algorithms and Computation, Lecture Notes in Computer Science, Vol. 4317, Kolkata, India, December 2006, Springer-Verlag, pp. 26–35.

[6] T. Lenz, Simple reconstruction of non-simple curves and approximating the median in streams with constant storage, Ph.D. dissertation, Freie Universität Berlin, 2008.

[7] J. I. Munro and V. Raman, "Selection from read-only memory and sorting with minimum data movement," Theoretical Computer Science **165**, 311–323, 1996.

[8] O. Reingold, Undirected connectivity in log-space, J. ACM **55**, (2008), Article #17, 24 pp.

[9] M. I. Shamos and D. Hoey, Closest-point problems, Proc. 16th Ann. IEEE Sympos. Found. Comput. Sci. (1975), pp. 151–162.