Preprocessing Imprecise Points and Splitting Triangulations

Marc van Kreveld

Maarten Löffler

Joseph S.B. Mitchell

Technical Report UU-CS-2009-007
March 2009

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Preprocessing Imprecise Points and Splitting Triangulations*

Marc van Kreveld
marc@cs.uu.nl

Maarten Löffler
loffler@cs.uu.nl

Joseph S.B. Mitchell
jsbm@ams.stonybrook.edu

Abstract

Traditional algorithms in computational geometry assume that the input points are given precisely. In practice, data is usually imprecise, but information about the imprecision is often available. In this context, we investigate what the value of this information is. We show here how to preprocess a set of disjoint regions in the plane of total complexity n in $O(n \log n)$ time so that if one point per set is specified with precise coordinates, a triangulation of the points can be computed in linear time. In our solution, we solve another problem which we believe to be of independent interest. Given a triangulation with red and blue vertices, we show how to compute a triangulation of only the blue vertices in linear time.

1 Introduction

Computational geometry deals with computing structure in 2-dimensional space (or higher). The most popular input is a set of points, on which something useful is then computed, for example, a triangulation. Algorithms for these tasks have been developed many years ago, and are provably fast and correct. However, there is one major obstacle to the practical application of geometric algorithms. Such algorithms work provably correct on the given input, but in practice, this input is often taken from the real world, and therefore has an intrinsic error. When the input is not precise, the value of the output is questionable.

In many applications, even though data is imprecise, some information about the error is known. For example, a point may be known not to be more than some ε away from a given point, to be inside a given region, or even to be chosen from a known probability distribution. Over the years, several approaches have been proposed to use this additional information, varying from fuzzy methods to computing partial output that is certain to be combinatorially correct.

1.1 Related work

Data imprecision in computational geometry has traditionally been considered mostly in stochastic or fuzzy settings [12, 20]. However, in recent years there has been a growing interest in exact models of imprecision. Guibas *et al.* [10] introduce the notion of *epsilon geometry*, a framework for robust computations on imprecise points. Abellanas *et al.* [1] and Weller [22] study the *tolerance* of a geometric structure: the largest perturbation of the vertices such that the combinatorial structure remains the same. Bandyopadhyay and Snoeyink [2] compute the set of "almost-Delaunay simplices", which are the tuples of points that can define a Delaunay simplex if the entire point

*This research was partially supported by the Netherlands Organisation for Scientific Research (NWO) through the project GOGO and under BRICKS/FOCUS grant no. 642.065.503. Partially supported by the National Science Foundation (CCF-0528209, CCF-0729019), Metron Aviation, and NASA Ames. A priliminary version of this research appeared at the 19th International Symposium on Algorithms and Computation (ISAAC 2008).

set is perturbed by at most $\varepsilon > 0$. Ely and Leclerc [9] and Khanban and Edalat [14] consider the epsilon geometry versions of the In-Circle predicate for Delaunay triangulation with imprecise points modeled as disks or rectangles, respectively. Khanban and co-authors [13, 15] developed a theory for returning partial Delaunay or Voronoi diagrams, consisting of the portion of the diagram that is certain. Van Kreveld and Löffler [18, 17] consider the problem of determining the smallest and largest possible values for geometric extent measures—such as the diameter or convex hull area—of a set of imprecise points.

1.2 Preprocessing imprecise points

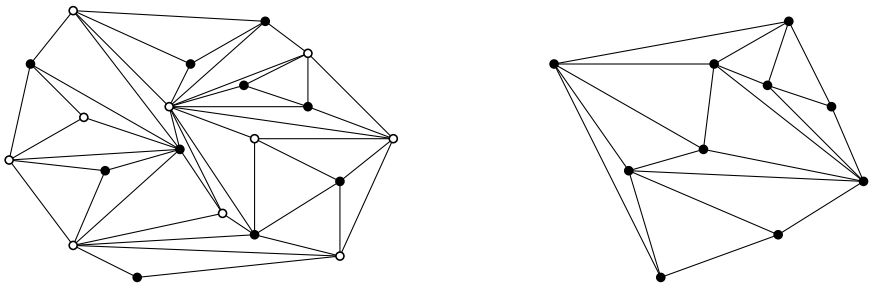
Here we study the situation where each point is known to lie in a predescribed region in the plane. We are interested in preprocessing such a collection of regions, such that if the points are later given precisely, we can do certain computations faster. This is not always possible: if all regions have a common intersection, then we could still get any point set as a precise sample, and lower bounds for the classical case still apply.

Some results in this model are already known. Held and Mitchell [11] consider the problem of preprocessing a set of n disjoint unit discs in $O(n \log n)$ time, such that when one point in each disc is given, the point set can be triangulated in linear time. They give a simple and practical solution. Their result can be extended to overlapping regions of different shapes, as long as the regions do not overlap more than a constant number of other regions, the regions are fat, and the sizes do not vary by more than a constant. In the same setting, Löffler and Snoeyink [16] show that such a set of discs can be preprocessed in $O(n \log n)$ time such that the Delaunay triangulation of the points can be computed in linear time. This algorithm relies on the linear-time constrained Delaunay triangulation algorithm for polygons by Chin and Wang [6], which would not be easy to implement. The same extensions to partially overlapping fat regions are possible.

In both previous results, the extensions work only up to a class of shapes bounded by a number of parameters, which end up as constant factors in the running times. An interesting question is what can be done for more general regions. As mentioned earlier, we cannot hope to do any useful preprocessing if the overlap of the regions is not bounded. However, for a set of n general disjoint regions in the plane, there is hope. For such a set of regions, the Delaunay triangulation is out of reach: Djidjev and Lingas [7] show that even if the sorted order of a set of points is given, computing the Delaunay triangulation has a $\Theta(n \log n)$ lower bound. If our set of regions is a set of vertical lines, then all information a preprocessing phase could compute is exactly this order (and the distances, but they can be computed from the order in linear time anyway). Here we show that, although the Delaunay triangulation is out of reach, we *can* preprocess a set of disjoint regions such that *some* triangulation of a sample can be computed in linear time. Also, our algorithm is simple and, in particular, does not depend on linear time polygon triangulation.

To solve the problem, we use a relatively new technique that is called *scaffolding*. We first build a *scaffold* (in our case a triangulation) that captures the typical layout of the points we really want to triangulate. Then, once they are known, we insert the points into the scaffold and then remove the scaffold using a *splitting* algorithm (also called a *hereditary* algorithm). The problem in this case is to split a triangulation: given a triangulation in the plane with red and blue vertices, we want to compute a triangulation of only the blue (or red) vertices in linear time. Splitting algorithms have been studied before. For example, Chazelle *et al.* [4] show how to compute, given a Delaunay triangulation with red and blue vertices, the Delaunay triangulation of the blue vertices in linear time. Related to this, Chan [3] shows how to compute the convex hull of a subset of the vertices of a simple polygon in linear time, and a triangulation in $O(n \log^* n)$ time. Our result improves this to linear time. Chazelle and Mulzer [5] show how to split a 3D convex hull.

In the next section, we study the triangulation splitting problem. Then, in Section 3, we show how to preprocess a set of disjoint regions in the plane for linear-time triangulation of sample points, using this result. In Section 4, some concluding remarks are given.



(a) (b)

Figure 1: (a) Example input with red (open) and blue (solid) points. (b) Example output.

2 The partial triangulation problem

Problem 1 *Given a triangulation embedded in the plane with vertices that are coloured either red or blue, compute a triangulation of only the blue vertices.*

Figure 1 shows an example of this problem.

To solve the problem, we will remove all of the red points one by one, until we only have blue points left. During this process, we will maintain a subdivision of the plane with certain properties, which allow us to quickly find new red points to remove and to remove them efficiently. We first describe this subdivision and some operations we can perform on it, and then give the algorithm and time analysis.

2.1 Structure and operations

During the algorithm, we will maintain a subdivision of the plane that uses the blue points and remaining red points as vertices. The subdivision is a special kind of *pseudotriangulation*. A pseudotriangulation is a subdivision of a convex region into *pseudotriangles*: simple polygons with exactly three convex vertices. The three convex vertices are also called the *corners* of the pseudotriangle, and the three polygonal lines connecting each pair of corners are called the *sides* of the pseudotriangle. (Note that we don't require a pseudotriangulation to be 'pointy' or 'minimal'.)

In the pseudotriangulation that we maintain, we allow only two types of faces: triangles and *foxes*. A fox is a pseudotriangle that has only one side which is not a straight edge, and for which all vertices along this side are blue, and the one remaining vertex is red. We call the red vertex the *chin* of the fox, and the other two convex vertices the *ears*. Figure 2 shows an example of a fox. For each fox, we store the chain of concave blue vertices in a balanced binary tree. If a pseudotriangulation only has triangles and foxes as faces, we call it *happy*.

Note that our input triangulation is happy, since it only has normal triangles. Also note that if we manage to remove all red vertices and maintain a happy subdivision, we cannot have any foxes

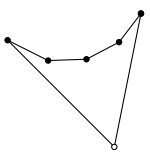
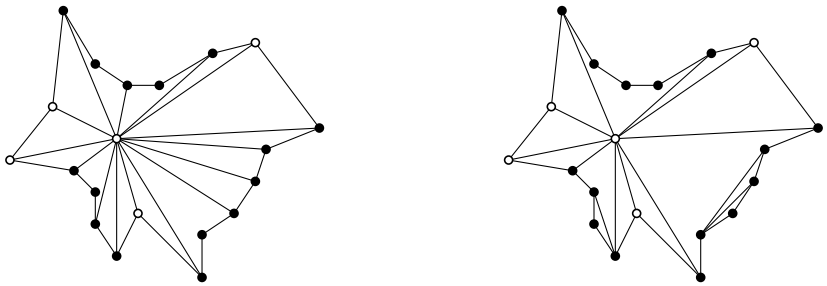


Figure 2: A fox is a pseudotriangle with one red vertex and one concave chain of blue vertices.



(a) (b)

Figure 3: (a) The pseudotriangles incident to a given red point form a star-shaped region. (b) By adding and removing the appropriate edges, we can make Condition $(*)$ hold.

left, since they have a red vertex: we are left with only normal triangles with three blue vertices, which is the required output of the algorithm.

Whenever we have a happy subdivision, we will denote the number of blue points by n and the number of remaining red points by k.

For a given red point p, let $r(p)$ be the number of red neighbours of p and $b(p)$ the number of blue neighbours of p (i.e., $r(p) + b(p)$ is the degree of p). Observe that any face which p is incident to is either a triangle or a fox that has p as its chin. As a consequence, the union of all faces incident to p forms a star-shaped polygon. By $c(p)$ we denote the total complexity of this polygon.

In addition to the shape restriction on the pseudotriangles, we will pose one more condition that we will maintain throughout the algorithm. For all red points p, Condition $(*)$ should hold.

$$b(p) \leq 2 \cdot r(p) + 3 \qquad (*)$$

This condition is not necessarily true for the input triangulation, so we will have to do an initial pass over the input triangulation to make this condition hold.

We will define some useful operations that we can perform on this triangulation.

2.1.1 Simplifying a red point

Lemma 1 *Let p be a red point in a happy pseudotriangulation. We can make Condition $(*)$ hold for p in $O(c(p))$ time.*

Figure 3 shows an example. Since p is a red point, and the pseudotriangulation is happy, the region around p (the union of its incident cells) is a star-shaped polygon of which all red and all convex vertices are connected to p. The purpose of this step is to remove any superfluous red-blue edges that p has. We leave all red-red edges where they are, so we don't need to consider them. Between each pair of red neighbours of p, there is a sector of the star that has only blue points (we will consider the case where p has no red neighbours separately). We will first prove the following lemma for a single sector.

Lemma 2 *Let p be a red point, and let q_1 and q_2 be two red neighbours of p such that there are no other red neighbours of p between them. Let b be the number of blue neighbours of p between q_1 and q_2, and c the total number of blue points between q_1 and q_2. In $O(b \log c + m)$ time or in $O(c)$ time, we can update the subdivision such that the number of red-blue edges from p to a point between q_1 and q_2 is at most 2 if $\measuredangle q_1 p q_2 \leq 180°$, and at most 3 otherwise.*

Proof: We have a sequence of blue points, some of which may be connected to p. The first and last points are always connected to p, since their neighbours are red, and any face of the subdivision with two red vertices must be a normal triangle. Now, if any other point s is also connected to p, we can almost always remove it. There are two cases.

If the angle at s after removing edge ps is concave, we can simply remove the edge. Both neighbours of s are blue, so the two cells ps separates must be foxes (or triangles with one red and two blue vertices, which are degenerate foxes). Since s is also concave, the combination of both cells is still a valid cell. In this case, we do need to concatenate the two binary trees that store the chains between the ears of the foxes. We postpone this concatenation until the end of the procedure.

If the angle at s after removing edge ps is convex, we can still remove the edge if the triangle formed by s and its two neighbours is empty. If this is the case, we add the edge between the neighbours of s, forming a completely blue triangle, then add edges from the two neighbours of s to p, and recurse. If the triangle is *not* empty, then we must keep ps. However, this is only possible if p itself is inside this triangle, which can happen at most once and only if the angle of the sector is at least 180°.

After all superfluous red-blue edges have been removed, we might be left with a sequence of $O(b)$ blue chains, stored as balanced binary trees, of total complexity $O(c)$, which have to be concatenated into one big balanced binary tree. We note that this can be done in $O(b \log c)$ time by merging them one by one, or in $O(c)$ time by simpy building a new tree from scratch. ⊠

With this result, we can prove Lemma 1.

Proof: We apply Lemma 2 to all sectors, using the $O(c)$ time complexity algorithm. There can be at most 2 sectors with an angle of at least 180°, so if p has any red neighbours the number of red-blue edges after simplifying all sectors is at most $2r(p) + 2$.

If p does not have any red neighbours, we can still proceed with deleting blue edges as described above, until there are only three neighbours left. In fact, in this case we are just triangulating the star-shaped polygon that remains after taking p out.

In both cases, Condition $(*)$ follows. ⊠

2.1.2 Subdividing a pseudotriangle

Lemma 3 *Let T be a pseudotriangle with the property that all concave vertices are blue. We can subdivide T into $O(1)$ non-blue triangles and foxes plus some number of triangles that are completely blue, in time $O(\log c + m)$ where c is the complexity of T and m is the number of blue-blue edges we produce in this step.*

Proof: If none of the sides of T have any concave vertices, then T is already a normal triangle.

If only one of the sides has concave vertices, and the corner opposite to it is blue, then we can triangulate the pseudotriangle with edges from the blue corner to all of the concave vertices, see Figure 4(a). This creates many blue triangles, and at most two triangles that involve a red point. If the corner opposite to it is red, then depending on the colours of the other two corners we either make an edge to the neighbours or not, see Figure 4(b). In this case, we make one fox and at most two triangles on the sides.

If two of the sides of the pseudotriangle have a concave vertex on them, consider the corner between these two sides. We can add an edge between its two neighbours, which are both blue. We can then continue adding blue-blue edges between the two chains, until this is no longer possible. The part that is left is then either a quadrilateral, see Figure 4(c), which we can simply split into two triangles, or a pseudotriangle with at most one side with concave vertices, see Figure 4(d), which we can further split in the way described above.

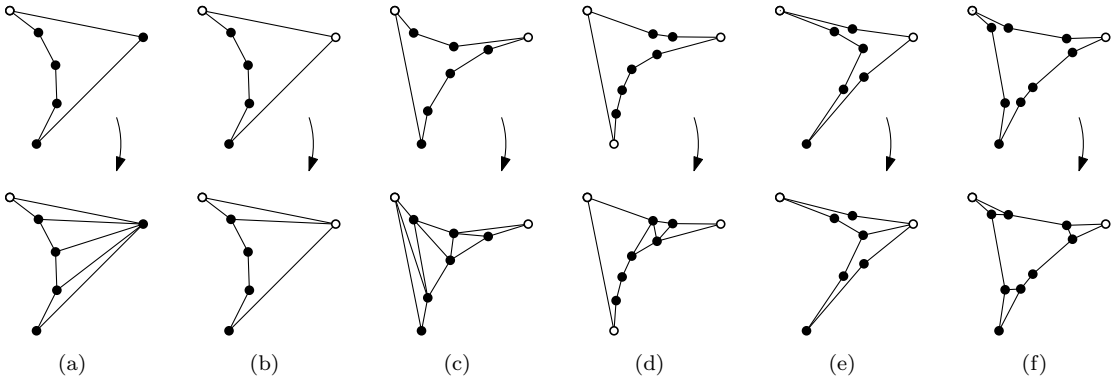


Figure 4: We can subdivide any pseudotriangle into $O(1)$ triangles and foxes, plus a number of polygons that only have blue vertices.

If all three sides have concave vertices on them, consider one of the corners. If we can make an edge between this corner and one concave vertex of the opposite side, then this splits the pseudotriangle into two pseudotriangles with both at most two sides with concave vertices, see Figure 4(e), which we can then further split as described above. We can find out whether there is such an edge in $O(\log c)$ time by extending the edges adjacent to the corner, and intersecting them with the opposite chain. If this is not possible for any corner, then we can connect the two neighbours of each corner with a blue-blue edge, see Figure 4(f). The remaining area has only blue vertices, and can be triangulated in any way. ⊠

2.1.3 Removing a red point

Lemma 4 *Let p be a red point in a happy subdivision with $r(p) = O(1)$, for which Condition $(*)$ holds. We can remove this point from the subdivision, and partition the gap it leaves into triangles and foxes in $O(\log c(p) + m)$ time, where m is the number of blue-blue edges formed in this step.*

Proof: Because of Condition $(*)$, we know that also $b(p) = O(1)$. We can remove p and all its incident edges, of which there are only a constant number. This results in an empty star-shaped polygon which needs to be partitioned into smaller cells again, see Figure 5(b). The complexity of this polygon is $c(p)$, and consists of $r(p)$ red points and at most $b(p)$ concave chains of blue points.

We will partition the gap into pseudotriangles. As described in [21], we can add geodesic shortest

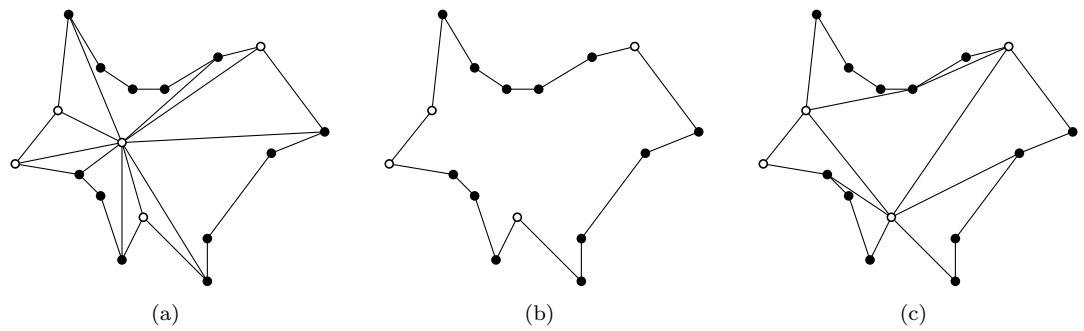


Figure 5: (a) A red point p of constant red degree and its incident pseudotriangles. (b) The empty polygon after removing p. (c) A repseudotriangulation of the gap.

paths between pairs of convex vertices of a simple polygon, until a (minimal) pseudotriangulation has been found. Since we have only a constant number of convex vertices, we only need to insert a constant number of shortest paths. For a given pair of vertices, we can compute this shortest path in $O(\log c(p))$ time, because we stored the chains of concave vertices in binary trees and we can compute tangents in logarithmic time. After this procedure, the gap has been split into a constant number of pseudotriangles in $O(\log c(p))$ time, see Figure 5(c). All the pseudotriangles have only blue concave vertices. We may have to split this binary trees that store the blue vertices into smaller parts, but only a constant number. One split can be done in logarithmic time, so this also takes $O(\log c(p))$ for all trees together.

We now apply Lemma 3 to all of these pseudotriangles to obtain a partitioning of the gap into a constant number of triangles and foxes, plus any necessary number of completely blue triangles. ⊠

We now have a happy subdivision again, although Condition $(*)$ may no longer be true for some red vertices on the boundary of the gap.

2.2 The algorithm

First, we must make sure that all red points in the pseudotriangulation satisfy Condition $(*)$. To this end, we simply apply Lemma 1 to all red points. This clearly takes linear time in total.

Now, we perform a sequence of reduction steps, each time reducing the number of red points by removing a constant fraction.

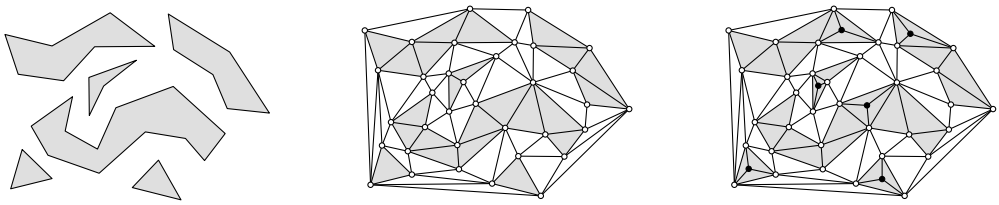
When we have k red points left, we want to find an independent set of $\Theta(k)$ red points that all have constant red degree. Since this step does not depend on any blue points, and because of Condition $(*)$, this can easily be done in $O(k)$ time.

Then, we remove each of those points by applying Lemma 4. The resulting subdivision is still happy, but Condition $(*)$ may not hold anymore for red points that had a red neighbour that was removed. However, we can repair this condition by applying Lemma 2 to those red points, but only in the sectors where something changed. The number of red-blue edges in those sectors cannot have increased by more than the number of edges that were added in the removal step. We added no more than a constant number of red-blue edges for each removed point, so this is in total at most $O(k)$.

2.3 Time analysis

The first phase takes $O(n)$ time.

Then, let $1/f$ denote the fraction of the red points that remain after throwing some away in each step (so $f > 1$). Then we perform $\log_f n$ phases, with at the ith phase $k = n/f^i$ red points left. In each phase, we spend $O(k)$ time to find an independent set. Then, we spend $O(\log c(p) + m)$ time for each element in the set. We can charge the m to the blue-blue edges that are created; since there can be at most $O(n)$ blue-blue edges and they are never removed, we spend no more than $O(n)$ time in total on them. The $c(p)$ terms are added over all elements in the independent set, and can be no more than $O(n)$ in total: $\sum_p c(p) = O(n)$. In the worst case, they are divided equally, and we spend $O(k \cdot \log \frac{n}{k})$ time on removing the points. By Lemma 2, Condition $(*)$ can be repaired in a sector that was involved in a removal step in $O(\log c(p) + m)$ time, since the number of red-blue edges in such a sector is constant. There are at most $O(k)$ such sectors, so again, in the worst case all blue points are equally divided and we spend $O(k \cdot \log \frac{n}{k})$ time on repairing them.



(a) (b) (c)

Figure 6: (a) A set of regions in the plane. (b) A triangulation of the vertices of the regions. (c) The sample points have been added to the triangulation.

The total time we spend is now:

$$\sum_{i=1}^{\log_f n} O(\frac{n}{f^i} \log f^i) = \sum_{i=1}^{\log_f n} O(f^i \log \frac{n}{f^i}) = \sum_{k=1}^{n} O(\log \frac{n}{u(k)})$$

where $u(k)$ is the smallest power of f larger than k. We can interpret this as the summation over k of the amount of time charged to removing one red point when there are k left. This time bound is then bounded by:

$$\sum_{k=1}^{n} O(\log \frac{n}{k}) = O(\log \frac{n^n}{n!}) = O(n \log n - \log n!) = O(n)$$

3 Triangulating imprecise points

Problem 2 *Given a set of non-overlapping convex regions in the plane, preprocess them in such a way that when a point in each region is given, the convex hull of these points can be computed in linear time.*

With the triangulation splitting result, it becomes straightforward to solve our original problem.

3.1 Preprocessing

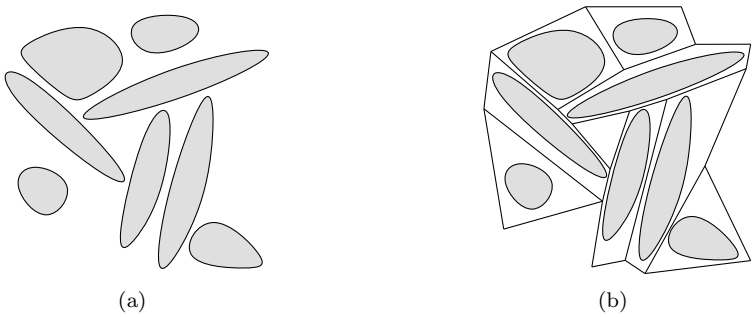
In preprocessing, we will compute a triangulation of the plane of complexity $O(n)$, such that each triangle only contains (a part of) one of the input regions. Together with this, we create a list of pointers from each imprecise region to the set of triangles that cover this region.

If the regions are disjoint polygons, as in Figure 6(a) (they need not be simple: they can have holes or multiple components), this is easy to do in $O(n \log n)$ time: we just compute a triangulation of the vertices of the regions, with the edges of the regions as required edges, as in Figure 6(b).

3.2 Reconstruction

Now, when we are given a set of points such that each point lies inside one of the input regions, we want to compute a triangulation of those points in linear time.

To do this, we add the points to the triangulation computed in the preprocessing step. For this we need to locate the points in the triangulation, which we do by simply walking through all triangles that this region points to. The point must lie in one of those triangles, and since each triangle is



(a) (b)

Figure 7: If the regions are not polygonal, we can find a polygonal subdivision of the plane such that each cell contains one region.

only pointed to once, we spend only linear time in total. Once the right triangle has been found, we simply split it into three smaller triangles. Figure 6(c) shows an example.

Now we have a triangulation with two types of vertices: the ones from the preprocessing (red) and the ones we added (blue). We simply invoke the algorithm from Section 2 to obtain a triangulation of just the blue points.

3.3 Extensions

The main improvement of our algorithm over [11] and [16] is that the input regions for the algorithm do not have to be fat or have the same size, or even be convex or connected. However, we did still assume that the regions are polygonal and do not overlap.

We can extend the approach to also work for regions that are not completely disjoint, as long as the complexity of their overlay is not too high. If we compute the overlay of the polygons and triangulate the resulting arrangement, the method will run in $O(n \log n + m)$ preprocessing and $O(mk \log k)$ reconstruction time, where m is the total complexity of the overlay, and k is the maximum number of regions that overlap in a single point.

If the input regions are not polygonal, we cannot simply triangulate their vertices, but the same approach still works if we first compute a polygonal subdivision of the plane such that each face contains one region. If the regions are convex, a subdivision exists which complexity is linear in the number of regions [8], see Figure 7. Such a subdivision can be computed in $O(n \log n)$ time [19]. If the regions are not convex, the complexity might increase, depending on the exact shape of the regions: for example, a region with a circular hole and another region which is a disc inside the hole may need an arbitrarily complex polygonal chain to separate.

4 Conclusions

When a set of points is unknown, but constrained by a known region for each point, it is interesting to preprocess the regions to speed up computations when the exact locations of the points become known. We give an algorithm to preprocess a set of disjoint regions in the plane in $O(n \log n)$ time, so that a sample from their regions can be triangulated in linear time. This time bound is optimal, and improves previous results by allowing more general regions. As future work, it would be interesting to study whether similar results can be obtained in higher dimensions.

As the main method of our solution, we use an algorithm for splitting a triangulation in linear time. This is a very natural problem that we believe is interesting in its own right, and which improves over an earlier $O(n \log^* n)$ algorithm.

References

[1] M. Abellanas, F. Hurtado, and P. A. Ramos. Structural tolerance and Delaunay triangulation. *Inf. Proc. Lett.*, 71:221–227, 1999.

[2] D. Bandyopadhyay and J. Snoeyink. Almost-Delaunay simplices: Nearest neighbour relations for imprecise points. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms*, pages 410–419, 2004.

[3] T. M. Chan. Three problems about simple polygons. *Comput. Geom. Theory Appl.*, 35(3):209–217, 2006.

[4] B. Chazelle, O. Devillers, F. Hurtado, M. Mora, V. Sacristán, and M. Teillaud. Splitting a Delaunay triangulation in linear time. *Algorithmica*, 34:39–46, 2002.

[5] B. Chazelle and W. Mulzer. Computing hereditary convex structures. In *Proc. 25th Symposium on Computational Geometry*, 2009 (to appear).

[6] F. Y. L. Chin and C. A. Wang. Finding the constrained Delaunay triangulation and constrained Voronoi diagram of a simple polygon in linear time. *SIAM J. Comput.*, 28(2):471–486, 1998.

[7] H. Djidjev and A. Lingas. On computing Voronoi diagrams for sorted point sets. *Internat. J. Comput. Geom. Appl.*, 5:327–337, 1995.

[8] H. Edelsbrunner, A. D. Robison, and X. Shen. Covering convex sets with non-overlapping polygons. *Discrete Math.*, 81:153–164, 1990.

[9] J. S. Ely and A. P. Leclerc. Correct Delaunay triangulation in the presence of inexact inputs and arithmetic. *Reliable Computing*, 6:23–38, 2000.

[10] L. J. Guibas, D. Salesin, and J. Stolfi. Constructing strongly convex approximate hulls with inaccurate primitives. *Algorithmica*, 9:534–560, 1993.

[11] M. Held and J. S. Mitchell. Input-constrained geometry. Submitted.

[12] W. S. Kendall, O. Barndorff-Nielson, and M. C. van Lieshout. *Current Trends in Stochastic Geometry: Likelihood and Computation.* Boca Raton, FL: CRC Press, 1998.

[13] A. A. Khanban. *Basic Algorithms of Computational Geometry with Imprecise Input.* PhD thesis, Imperial College, London, 2005.

[14] A. A. Khanban and A. Edalat. Computing Delaunay triangulation with imprecise input data. In *Proc. 15th Canad. Conf. on Comput. Geom.*, pages 94–97, 2003.

[15] A. A. Khanban, A. Edalat, and A. Lieutier. Computability of partial Delaunay triangulation and Voronoi diagram. In V. Brattka, M. Schröder, and K. Weihrauch, editors, *Electronic Notes in Theoretical Computer Science*, volume 66. Elsevier, 2002.

[16] M. Löffler and J. Snoeyink. Delaunay triangulations of imprecise points in linear time after preprocessing. In *Proc. 24th Symposium on Computational Geometry*, pages 298–304, 2008.

[17] M. Löffler and M. van Kreveld. Largest bounding box, smallest diameter, and related problems on imprecise points. In *Proc. 10th Workshop on Algorithms and Data Structures*, LNCS 4619, pages 447–458, 2007.

[18] M. Löffler and M. van Kreveld. Largest and smallest convex hulls for imprecise points. *Algorithmica*, 2008. doi:10.1007/s00453-008-9174-2, in press.

[19] M. Pocchiola and G. Vegter. On polygonal covers. In B. Chazelle, J. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 257–268. AMS, Providence, 1999.

[20] A. Rosenfeld. Fuzzy geometry: An updated overview. *Inf. Sci.*, 110(3-4):127–133, 1998.

[21] G. Rote, C. A. Wang, L. Wang, and Y. Xu. On constrained minimum pseudotriangulations. In *Proc. 9th International Computing and Combinatorics Conference*, LNCS 2697, pages 445–454, 2003.

[22] F. Weller. Stability of Voronoi neighborship under perturbations of the sites. In *Proc. 9th Canad. Conf. Comput. Geom.*, pages 251–256, 1997.