Chapter 7

Reductions

A problem Q_1 is *reducible* to the problem Q_2 in time $O(t(n))$, or equivalently, the problem Q_1 is $t(n)$-time reducible to Q_2, written as

$$Q_1 \propto_{t(n)} Q_2,$$

if there are a *forward algorithm A_f* and a *backward algorithm A_b* such that

1. For an instance x_1 of the problem Q_1, the forward algorithm A_f converts x_1 in time $O(t(|x_1|))$ to an instance x_2 of the problem Q_1;
2. On a solution y_2 to the instance x_2 of Q_2, the backward algorithm A_b converts y_2 in time $O(t(|x_1|))$ to a solution y_1 to the instance x_1 of Q_1.

Note that in the above definition, there is no assumption on how difficult the problem Q_2 is. Moreover, the running time of the backward algorithm A_b is measured in terms of the size of the instance x_1 of the problem Q_1. The following theorem shows that if the problem Q_2 can be solved efficiently, then the problem Q_1 can also be solved efficiently.

Lemma 7.0.1 *Suppose that a problem Q_1 is $t(n)$-time reducible to a problem Q_2, i.e., $Q_1 \propto_{t(n)} Q_2$, and that the problem Q_2 can be solved in time $O(T(n))$. Then the problem Q_1 can be solved in time $O(t(n) + T(O(t(n))))$.*

PROOF. Suppose that a $t(n)$-time reduction from the problem Q_1 to the problem Q_2 is given by a forward algorithm A_f and a backward algorithm A_b, and that an algorithm A_2 solves the problem Q_2 in time $O(T(n))$. The problem Q_1 can be solved as follows, on an instance x_1 of Q_1:

(1) use the forward algorithm A_f to convert x_1 to an intance x_2 of Q_2;
(2) use the algorithm A_2 on x_2 to construct a solution y_2 to x_2; and

(3) use the backward algorithm A_b to convert y_2 into a solution y_1 to the instance x_1 of Q_1.

Let $n = |x_1|$. By the definition, step (1) and step (3) of the above algorithm take time $O(t(n))$. Since step (1) takes time $O(t(n))$, the size of the instance x_2 of Q_2 is bounded by $O(t(n))$. Therefore, in step (2) of the algorithm, the algorithm A_2 of time complexity $O(T(m))$ on inputs of size m takes time $O(T(O(t(n))))$ on the input x_2. This concludes that the running time of the above algorithm that solves the problem Q_1 is bounded by

$$O(t(n)) + O(T(O(t(n)))) = O(t(n) + T(O(t(n)))). \qquad \square$$

The reduction technique plays an important role in the study of complexity of geometric problems, both for deriving lower bounds and for designing efficient algorithms. In this chapter, we study how to use the technique to design efficient geometric algorithms. In the next chapter, we will explain how we use the technique to derive lower bounds for geometric problems.

We close this introductory section by the following corollary, which will be heavily used in our discussion.

Corollary 7.0.2 *Suppose that problem Q_1 is linear-time reducible to problem Q_2, i.e., $Q_1 \propto_n Q_2$. If problem Q_2 can be solved in time $O(T(n))$, where the function $T(n)$ satisfies $T(n) = \Omega(n)$ and $T(O(n)) = O(T(n))$, then problem Q_1 can also be solved in time $O(T(n))$.*

PROOF. As shown in Lemma 7.0.1, the problem Q_1 can be solved by an algorithm A_1 in time $O(n + T(O(n)))$. By our assumption, $T(O(n)) = O(T(n))$. Moreover, $T(n) = \Omega(n)$. Therefore, the time complexity of the algorithm A_1 under these conditions is bounded by

$$O(n + O(T(n))) = O(T(n)). \qquad \square$$

Notice that most of the complexity functions $T(n)$ we use in this book, such as n, $n \log n$, n^k, and $n^k \log^h n$, where k and h are fixed constants, satisfy the conditions $T(n) = \Omega(n)$ and $T(O(n)) = O(T(n))$.

7.1 Convex hull and sorting

Consider the algorithm `Modified-Graham-Scan` (subsection 4.2.2) for constructing the convex hull of a set S of points in the plane. If the given set S of n points in the plane is sorted by x-coordinates, then the algorithm

`Modified-Graham-Scan` needs only linear time to construct the convex hull CH(S) for S. In fact, it is not hard to see that

$$\text{Convex-Hull} \propto_n \text{Sorting}$$

by the following argument. Given an instance of Convex-Hull, which is a set S of n points in the plane, we can take S as an instance of Sorting if we let the x-coordinate of a point p in S be the "key" of the point p. Therefore, we can simply pass the instance S of the problem Convex-Hull to the problem Sorting as an instance S. Now the solution of Sorting on the instance S is a list of the points in S that is sorted by the x-coordinates. The `Modified-Graham-Scan` algorithm shows that with this solution to Sorting, the convex hull CH(S) of the set S, which is the solution to the instance S of Convex-Hull, can be constructed in time $O(n)$.

It is interesting that we can prove that the problem Sorting can also be reduced to the problem Convex-Hull in linear time.

Theorem 7.1.1 *Sorting $\propto_n$ Convex-Hull.*

Proof. Given a set $L = (x_1, x_2, \ldots, x_n)$ of n real numbers, which is an instance of the problem Sorting, we can suppose that all numbers in the set L are non-negative – otherwise, we first scan the set L and find the smallest number x in L, then add $-x$ to each number in L to produce a new set L' of non-negative numbers. From a sorted list of the new set L', we can easily get a sorted list of the original set L.

We first scan the set L to find the largest number $x_{\max}$ in L. Then for each number x_i in L, we convert x_i into a point p_i in the plane such that the polar angle of p_i is $2\pi x_i/x_{\max}$ and the distance between p_i and the origin O is 1 (so the point p_i is on the unit circle). Let S be the set of these n points p_1, p_2, …, p_n in the plane. The set S is an instance of the problem Convex-Hull, which can obviously be obtained from the set L in time $O(n)$, since all we do is to scan the set L to find the largest number $x_{\max}$, and for each number x_i in L, in constant time to construct the point p_i.

Since the unit circle is convex, the n points in the set S must be all on the convex hull CH(S). Therefore, the solution CH(S) to the instance S of the problem Convex-Hull is a list $H_L = \langle p_{i_1}, p_{i_2}, \ldots, p_{i_n} \rangle$ of the points in S, given in the order when we traverse the convex hull CH(S) in counterclockwise ordering. Scanning the points in the list H_L for CH(S), we can find the point in S that has the minimum polar angle (by circularly rotating the list H_L, we can assume without loss of generality that point p_{i_1} has the minimum polar angle). Thus, the list H_L is given in increasing order of the polar angles of the points in S. Since the polar angle of the point p_i in the set S is $2\pi x_i/x_{\max}$, where x_i is the corresponding number in the list L, the ordered list $L' = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_n} \rangle$ obtained by multiplying the polar angle of each point in the list H_L by $x_{\max}/(2\pi)$ gives the sorted list of the original list L, which is the solution to the instance L for the problem SORTING. Finally, it is easy to verify that the set S of the points, which is an instance of the problem CONVEX-HULL, can be constructed in time $O(n)$ from the set L of numbers, which is an instance of the problem SORTING, and that the sorted list L', which is the solution to the set L, can be constructed in time $O(n)$ from the list H_L, which represents the convex hull CH(S) that is the solution to S. In conclusion, this shows that the problem SORTING is n-time reducible to the problem CONVEX-HULL. $\square$

Let Q_1 and Q_2 be two problems and $t(n)$ be a function. If we have both

$$Q_1 \propto_{t(n)} Q_2 \qquad \text{and} \qquad Q_2 \propto_{t(n)} Q_1,$$

then we say that the problems Q_1 and Q_2 are *equally complex* up to a $t(n)$-time reduction, written as $Q_1 \equiv_{t(n)} Q_2$.

For two problems that are equally complex up to a linear time reduction, if one can be solved in time $O(T(n))$, where $T(n) = \Omega(n)$ and $T(O(n)) = O(T(n)$, then by Corollary 7.0.2, the other can also be solved in time $O(T(n))$.

By the above discussions, we have proved that SORTING $\equiv_n$ CONVEX-HULL. In fact, construction of convex hulls for sets of points in the plane is a generalization of sorting. In sorting n numbers, we are asked to find the ordering of a set of points in the real line, while in constructing a convex hull, we are asked to find the ordering of polar angles, relative to an interior point of the convex hull, of the "extreme points". The difference is that in sorting, every given number appears in the final sorted list, while in constructing a convex hull, we also have to make the decision on whether a given point is a non-extreme point, and if yes, exclude it from the final output list. On the other hand, as we have seen, up to an linear-time reduction, sorting is not easier than constructing convex hulls for points in the plane.

7.2 Closest pair and all nearest neighbor

According to the definition of the Voronoi diagram (a partition of the plane into regions such that each region is the locus of points closer to a point of the set S than to any other point of S), it is not surprising that the problems CLOSEST-PAIR and ALL-NEAREST-NEIGHBORS can be solved efficiently through the Voronoi diagram. Recall that the problem CLOSEST-PAIR is to find the closest pair in a given set of n points in the plane, while the problem ALL-NEAREST-NEIGHBORS is that for each point in a given set S of n points in the plane, find the nearest neighbor in S. Finally, let VORONOI-DIAGRAM denote the problem of constructing the Voronoi diagram for a given set of n points in the plane.

Theorem 7.2.1 ALL-NEAREST-NEIGHBORS $\propto_n$ VORONOI-DIAGRAM.

PROOF. Suppose that a set S of n points in the plane is an input instance to the problem ALL-NEAREST-NEIGHBORS, we pass the input S directly as an instance to the VORONOI-DIAGRAM problem. The solution to the instance S of the problem VORONOI-DIAGRAM is the Voronoi diagram $\text{Vor}(S)$ of the set S. By Lemma 5.2.2, for each point p_i in the set S, the nearest neighbor of p_i in S defines a non-degenerate Voronoi edge for the Voronoi polygon V_i for p_i. Therefore, by traversing the boundary of the Voronoi polygon V_i for the point p_i, we can find the nearest neighbor in the set S for the point p_i. Doing this for all points in S gives the solution to the instance S of the problem ALL-NEAREST-NEIGHBORS. Given the Voronoi diagram $\text{Vor}(S)$ of the set S, each Voronoi polygon can be traversed by the algorithm `Trace-Region` given in section 2.4 in time proportional to the number of edges on the boundary of the polygon. Since each Voronoi edge is on the boundary of exactly two Voronoi polygons, the sum of the total numbers of boundary edges over all Voronoi polygons in $\text{Vor}(S)$ equals twice of the number of edges in the Voronoi diagram $\text{Vor}(S)$. As a result, we conclude that traversing all Voronoi polygons of the Voronoi diagram $\text{Vor}(S)$, thus finding the nearest neighbor for each point in the set S when the Voronoi diagram $\text{Vor}(S)$ is given, takes time proportional to the number of edges of $\text{Vor}(S)$, which is of order $O(n)$ since the Voronoi diagram is a planar graph. This proves the reduction ALL-NEAREST-NEIGHBORS $\propto_n$ VORONOI-DIAGRAM. $\square$

Since the Voronoi diagram of a set of n points can be constructed in time $O(n \log n)$ (see Theorem 5.3.7), by Corollary 7.0.2, we obtain

Corollary 7.2.2 *The problem* ALL-NEAREST-NEIGHBORS *can be solved in time* $O(n \log n)$.

It is easy to see that given a set S of n points in the plane, the solution to the instance S of the problem CLOSEST-PAIR can be obtained from the solution to the instance S of the problem ALL-NEAREST-NEIGHBORS in linear time, by simply computing the distance between each point and its nearest neighbor, then taking the point that has the shortest distance to its nearest neighbor. This gives

$$\text{CLOSEST-PAIR} \propto_n \text{ALL-NEAREST-NEIGHBORS}.$$

By Corollary 7.2.2, the problem ALL-NEAREST-NEIGHBORS can be solved in time $O(n \log n)$. Therefore by Corollary 7.0.2, we have

Corollary 7.2.3 *The problem* CLOSEST-PAIR *is sovable in time* $O(n \log n)$.

7.3 Triangulation

Let $\text{Vor}(S)$ be the Voronoi diagram for a set S of n points in the plane. We draw a segment $[p_i, p_j]$ for each pair of points p_i and p_j in S if they define a Voronoi edge in $\text{Vor}(S)$. Let $D(S)$ be the collection of these segments, which is called the *straight-line dual* of the Voronoi diagram $\text{Vor}(S)$.

We prove that the straight-line dual $D(S)$ of the Voronoi diagram $\text{Vor}(S)$ is a triangulation of the point set S, i.e., the straight-line dual $D(S)$ partitions the convex hull $\text{CH}(S)$ of the point set S into triangles such that

(1) no two of the triangles overlap in the interior, and
(2) every point in the convex hull $\text{CH}(S)$ (more precisely, every point in the area bounded by the convex hull $\text{CH}(S)$) must be contained in at least one of the triangles.

In the Voronoi diagram $\text{Vor}(S)$ of the point set S, under the assumption that no four points in the set S are co-circular, by Lemma 5.2.1, each Voronoi vertex v is incident to exactly three Voronoi edges e_1, e_2, and e_3, and to exactly three Voronoi polygons V_1, V_2, and V_3, which are for three points p_1, p_2, and p_3, respectively, in the set S. Each of the edges e_1, e_2, and e_3 is defined by a pair of the points p_1, p_2, and p_3. Therefore, the segments $[p_1, p_2]$, $[p_2, p_3]$, and $[p_3, p_1]$ are all in the straight-line dual $D(S)$ of $\text{Vor}(S)$ and form a triangle in $D(S)$. Thus, each Voronoi vertex v in $\text{Vor}(S)$ is associated with a triangle $\triangle p_1 p_2 p_3$ in the straight-line dual $D(S)$. Denote by $\triangle(v)$ the triangle $\triangle p_1 p_2 p_3$. On the other hand, since a Voronoi edge $[v, v']$ in $\text{Vor}(S)$ is incident on two Voronoi vertices v and v', the segment in the straight-line dual $D(S)$ corresponding to the edge $[v, v']$ is a boundary edge of the two triangles $\triangle(v)$ and $\triangle(v')$ in $D(S)$.

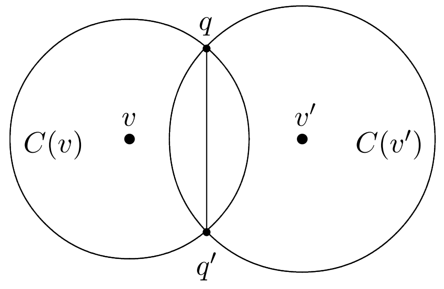

Figure 7.1: Two circumcircles intersect at q and q'

Lemma 7.3.1 *No two triangles $\triangle(v)$ and $\triangle(v')$ in $D(S)$ overlap in the interior, where v and v' are two Voronoi vertices in Vor(S).*

PROOF. Let $\triangle(v)$ and $\triangle(v')$ be two arbitrary triangles in the straight-line dual $D(S)$ of the Voronoi diagram Vor(S). Let $C(v)$ and $C(v')$ be the circumcircles of the triangles $\triangle(v)$ and $\triangle(v')$, respectively. Thus, the triangle $\triangle(v)$ is entirely contained in $C(v)$ while the triangle $\triangle(v')$ is entirely contained in $C(v')$. If the circumcircles $C(v)$ and $C(v')$ do not overlap in the interior, then of course the triangles $\triangle(v)$ and $\triangle(v')$ do not overlap in the interior. Thus, we assume that $C(v)$ and $C(v')$ do overlap in the interior. Note that (under the assumption of no four co-circular points in S) each of the circumcircles $C(v)$ and $C(v')$ contains exactly three points in the point set S on its boundary, and by Lemma 5.2.3, no point of S is contained in the interior of $C(v)$ or $C(v')$. Moreover, $C(v)$ and $C(v')$ cannot be coincide – otherwise at least four points in the set S would be co-circular. Also, no one of the circles $C(v)$ and $C(v')$ can be entirely contained in the other – otherwise some point of the set S would be contained in the interior of $C(v)$ or $C(v')$, contradicting to Lemma 5.2.3. So the boundaries of the circumcircles $C(v)$ and $C(v')$ must intersect at exactly two points q and q'. See Figure 7.1.

The two points q and q' partition the circle $C(v)$ into two disjoint curves, one is entirely contained in the circle $C(v')$ and the other is completely outside the circle $C(v')$. No vertex of the triangle $\triangle(v)$ can be on the curve of $C(v)$ that is entirely contained in the circle $C(v')$ – otherwise the vertex of $\triangle(v)$, which is a point in the point set S, would be in the interior of the circle $C(v')$, contradicting Lemma 5.2.3. Thus the three vertices of $\triangle(v)$ must be on the curve of $C(v)$ that is outside $C(v')$. Similarly, the three vertices of the triangle $\triangle(v')$ are on the curve of $C(v')$ that is outside $C(v)$. As a

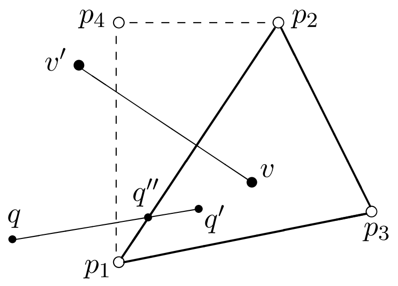

Figure 7.2: A point q outside all triangles

consequence, the three vertices of the triangle $\triangle(v)$ and the three vertices of the triangle $\triangle(v')$ must be separated by the segment $[q, q']$, so the triangles $\triangle(v)$ and $\triangle(v')$ cannot overlap in the interior. $\square$

Every segment in the straight-line dual $D(S)$ is an edge for some triangles in $D(S)$. Moreover, by Lemma 5.2.3, no point p in S can be an interior point in a segment in $D(S)$ (otherwise, the point p would be in the interior of the circumcircle $C(v)$ for some Voronoi vertex v). Thus, Lemma 7.3.1 implies that the segments in $D(S)$ can only intersect at their ends. Therefore, the straight-line dual $D(S)$ of Vor(S) is a PSLG. To show that $D(S)$ is a triangulation of the point set S, we also need to prove the following lemma.

Lemma 7.3.2 *Every point in the convex hull CH(S) is contained in a triangle $\triangle(v)$ for some Voronoi vertex v in the Voronoi diagram Vor(S).*

PROOF. Suppose that the lemma is not true and that some point q in the convex hull CH(S) is not contained in any such triangles. Then we can find a triangle $\triangle(v)$, where v is a Voronoi vertex of Vor(S), and an interior point q' in the triangle $\triangle(v)$ such that the segment $[q, q']$ intersects no triangles in $D(S)$ except the triangle $\triangle(v)$. Moreover, we can suppose that the segment $[q, q']$ intersects the boundary of the triangle $\triangle(v)$ at a point that is not a vertex of $\triangle(v)$. This condition can always be satisfied since we can move the point q', which is an interior point in $\triangle(v)$, slightly in the triangle $\triangle(v)$.

Therefore, we can suppose that the three vertices of the triangle $\triangle(v)$ are p_1, p_2, and p_3, which are points in the set S, that the segment $[q, q']$ intersects the edge $[p_1, p_2]$ of the triangle $\triangle(v)$ at an internal point q'', and that no point on the segment $[q, q'']$ (excluding the point q'') is contained in any triangle $\triangle(u)$ for a Voronoi vertex u. See Figure 7.2. Under these conditions, the point p_3 and the point q must be on different sides of the segment $[p_1, p_2]$. Since both points q and p_3 are contained in the convex hull CH(S), the segment $[p_1, p_2]$ cannot be a boundary edge of CH(S). Let $e = [v, v']$ be the Voronoi edge defined by the points p_1 and p_2 in S (note that the vertex v must be an end-point of e), then by Lemma 5.2.4, e is a finite edge since the points p_1 and p_2 are not consecutive hull vertices on CH(S). So v' is a finite Voronoi vertex in Vor(S), and defines a triangle $\triangle(v')$ in the straight-line dual $D(S)$. By the definition of $\triangle(v')$, two vertices of $\triangle(v')$ must be the points p_1 and p_2, and the other vertex p_4 of $\triangle(v')$ must be different from the point p_3 since $v \neq v'$. By Lemma 7.3.1, the two triangles $\triangle(v)$ and $\triangle(v')$ do not overlap in the interior, so the two points p_3 and p_4 must be on different sides of the segment $[p_1p_2]$ (see Figure 7.2). Consequently, however, some points on the segment $[q, q'']$ that are close enough to the point q'' would have to be contained in the interior of the triangle $\triangle(v')$. This contradicts our assumption that no points on $[q, q'']$ (excluding q'') is contained in any triangles in $D(S)$. This contradiction shows that q must belong to a triangle $\triangle(w)$ in $D(S)$ for some Voronoi vertex w in *Vor(S)*. $\square$

By Lemma 7.3.1 and Lemma 7.3.2, we conclude immediately that the straight-line dual $D(S)$ of the Voronoi diagram Vor(S) is a triangulation of the point set S. This triangulation of the point set S is called the *Delaunay triangulation* of the set S.

Define TRIANGULATION as the problem of constructing a triangulation for a given point set S in the plane such that all points in the convex hull CH(S) of S are contained in the triangles, and let VORONOI-DIAGRAM be the problem of constructing the Voronoi diagram Vor(S) for a given point set S in the plane. Then we have

Theorem 7.3.3 TRIANGULATION $\propto_n$ VORONOI-DIAGRAM.

PROOF. Let S be a set of n points in the plane, which is an instance of the problem TRIANGULATION. We simply pass S as an instance to the problem VORONOI-DIAGRAM. The solution to the instance S of the problem VORONOI-DIAGRAM is the Voronoi diagram Vor(S) of the point set S. From the Voronoi diagram Vor(S), given in a doubly connected edge list (DCEL), we construct the Delaunay triangulation $D(S)$ of S in time $O(n)$ by traversing all the Voronoi vertices of Vor(S). Note that by traversing the edges incident to a Voronoi vertex v, we can easily construct the triangle $\triangle(v)$ in $D(S)$. Once all the triangles in $D(S)$ are constructed, we can easily construct the doubly connected edge list for $D(S)$ in time $O(n)$. Summarizing the discussion, we conclude that the TRIANGULATION problem is linear-time reducible to the VORONOI-DIAGRAM problem. □

By Theorem 5.3.7, the Voronoi diagram of a set of n points in the plane can be constructed in time $O(n \log n)$. By Corollary 7.0.2, we have

Corollary 7.3.4 *The problem* TRIANGULATION *can be solved in time $O(n \log n)$. In particular, the Delaunay triangulation $D(S)$ of a set S of n points in the plane can be constructed in time $O(n \log n)$.*

7.4 Euclidean minimum spanning tree

Consider the following problem: given a set S of n points in the plane, interconnect all the points by straight line segments so that the sum of the lengths of the segments is minimized. The problem has obvious applications in computer networking where we want to interconnect all the computers at the minimum cost. In fact, the problem also has wide applications in many other areas, such as bioinformatics and industrial managements.

It is easy to see that the resulting connected PSLG after the above interconnection must be a tree. In fact, if the resulting PSLG were not a tree, then we would be able to find a cycle in the PSLG. Deleting an edge from the cycle would still keep the PSLG connected. But this would contradict the assumption that the resulting connected PSLG has the minimum sum of the lengths of its segments over all such connected PSLGs. This tree is called a *Euclidean minimum spanning tree (EMST)* of the set S. In general, the Euclidean minimum spanning tree of a set is not unique.

The problem of finding a Euclidean minimum spanning tree for a set of points in the plane is closely related to the following problem of finding a *minimum weight spanning tree* in a graph: given a weighted graph G, find a spanning tree of G with the minimum total weight. In fact, the problem of finding a Euclidean minimum spanning tree can be reduced to the problem of finding a minimum weight spanning tree in a weighted graph, as follows.

Let S be a set of n points in the plane. To construct a Euclidean minimum spanning tree of S, we can regard S as a weighted complete graph G_S of n vertices, which are the n points in the set S, such that the weight of an edge $e = [p, p']$ in G_S is the Euclidean distance between the points p and p'. Therefore, a Euclidean minimum spanning tree of the set S is a minimum weighted spanning tree of the graph G_S, and vice versa. There are a few efficient algorithms for constructing the minimum weighted spanning tree for weighted graphs. For example, Kruskal's algorithm [16] constructs the minimum weighted spanning tree for a weighted graph with n vertices and m edges in time $O(m \log n)$. However, the complete graph G_S has $\binom{n}{2} = \Omega(n^2)$ edges. Therefore, a direct application of Kruskal's algorithm to the complete graph G_S would result in an $O(n^2 \log n)$-time algorithm for constructing a Euclidean minimum spanning tree for the set S. Another well-known for constructing the minimum weighted spanning tree for a weighted graph is Prim's algorithm [25], which runs in time $O(n^2)$.

One possible way to have a more efficient algorithm for the problem is to exclude most of the edges in the graph G_S. For example, if we can exclude all but $O(n)$ edges in the graph G_S, then Kruskal's algorithm on the graph G_S with fewer edges would give an $O(n \log n)$-time algorithm for the Euclidean minimum spanning tree problem. Interesting enough, this can be achieved with the help of the Voronoi diagram and the Delaunay triangulation of point sets in the plane. We start with a few lemmas on the relationship between the Delaunay triangulation and the Euclidean spanning trees.

Lemma 7.4.1 *Partition the point set S into two non-empty disjoint subsets S_1 and S_2. If $[p_1, p_2]$ is the shortest line segment with $p_1 \in S_1$ and $p_2 \in S_2$, then the segment $[p_1, p_2]$ is an edge in the Delaunay triangulation $D(S)$.*

Proof. Suppose that the segment $[p_1, p_2]$ is not an edge in the Delaunay triangulation $D(S)$. By the definition of $D(S)$, the perpendicular bisector of $[p_1, p_2]$ contains no Voronoi edge of $\mathrm{Vor}(S)$. Let V_1 be the Voronoi polygon for the point p_1 in the Voronoi diagram $\mathrm{Vor}(S)$, and suppose that the segment $[p_1, p_2]$ intersects the Voronoi polygon V_1 at a point q that is on the Voronoi edge e of the Voronoi polygon V_1 in $\mathrm{Vor}(S)$. Note that the point p_2 cannot be contained in V_1 (including the boundary of V_1) since the polygon V_1 is the locus of points closer to p_1 than to any other points in S. Suppose that the Voronoi edge e is defined by the point p_1 and another point p_3 in the set S. See Figure 7.3. By the definition, the points p_1 and p_3 are the closest points in the set S to the points on the edge e. Therefore (here $|pq|$ denotes the length of the segment $[p, q]$),

$$|p_1p_2| = |p_1q| + |qp_2| \geq |p_1q| + |qp_3| > |p_1p_3|, \tag{7.1}$$

where the last inequality is due to the fact that the point q cannot be on the segment $[p_1, p_3]$ – otherwise, the three points p_1, p_2 and p_3 would be co-linear. Moreover, since we have $\angle qp_3p_1 = \angle p_3p_1q$, and the point q is an

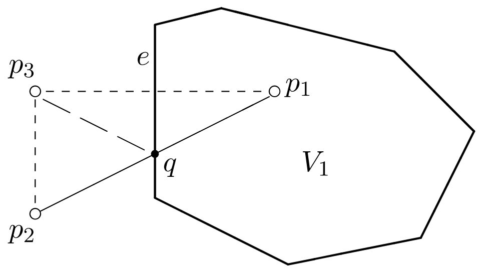

Figure 7.3: $[p_1, p_2]$ intersects V_1 at q

internal point of the segment $[p_1p_2]$, we must have

$$\angle p_2p_3p_1 > \angle qp_3p_1 = \angle p_3p_1q = \angle p_3p_1p_2$$

By elementary geometry (larger side is opposite larger angle), we have

$$|p_1p_2| > |p_2p_3|. \tag{7.2}$$

Now we obtain a contradiction. By inequalities (7.1) and (7.2), both segments $[p_1, p_3]$ and $[p_2, p_3]$ are shorter than the segment $[p_1, p_2]$. If $p_3 \in S_1$ then we pick the segment $[p_2, p_3]$, and if $p_3 \in S_2$ then we pick the segment $[p_1, p_3]$. No matter what set the point p_3 belongs to, we would always be able to find a segment with one end in S_1 and the other end in S_2 such that the segment is shorter than the segment $[p_1, p_2]$. This contradicts the assumption that $[p_1, p_2]$ is the shortest such a segment.

This contradiction proves that the segment $[p_1, p_2]$ must be an edge in the Delaunay Triangulation $D(S)$ of the point set S. $\square$

Lemma 7.4.2 *Let p_1 and p_2 be two points in the set S. The segment $[p_1, p_2]$ is an edge of a Euclidean minimum spanning tree for S if and only if there is a partition of the set S into two subsets S_1 and S_2 such that $[p_1, p_2]$ is the shortest segment with one end in S_1 and the other end in S_2.*

Proof. Suppose that $[p_1, p_2]$ is an edge of a Euclidean minimum spanning tree T for the set S. Deleting the edge $[p_1, p_2]$ from T results in two disjoint subtrees T_1 and T_2. Let S_1 and S_2 be the sets of points in S that are the vertices of the trees T_1 and T_2, respectively. The sets S_1 and S_2 obviously form a partition of the set S and each of the sets S_1 and S_2 contains exactly one of the points p_1 and p_2. We claim that the segment $[p_1, p_2]$ is the shortest segment with one end in S_1 and the other end in S_2. In fact, if

$[p, p']$ is a shorter segment with one end in S_1 and the other end in S_2, then the segment $[p, p']$ cannot be in the tree T – otherwise, the two subtrees T_1 and T_2 would be connected by the two segments $[p, p']$ and $[p_1, p_2]$, and the tree T would contain a cycle. Now in the tree T, replacing the segment $[p_1, p_2]$ by the segment $[p, p']$ would give a Euclidean spanning tree T' for S such that the sum of the edge lengths of the tree T' is strictly less than the sum of the edge lengths of the tree T, contradicting the assumption that T is a Euclidean minimum spanning tree for S. As a conclusion, $[p_1, p_2]$ must be the shortest segment with one end in S_1 and the other end in S_2.

Conversely, suppose that there is a partition of the point set S into two non-empty subsets S_1 and S_2 such that $[p_1, p_2]$ is the shortest segment with one end in S_1 and the other end in S_2. Let T be a Euclidean minimum spanning tree for the set S. If T contains $[p_1, p_2]$, then we are done. Otherwise, adding the segment $[p_1, p_2]$ to the tree T results in a unique simple cycle C. Since the segment $[p_1, p_2]$ is on the cycle C and p_1 and p_2 are in different sets of S_1 and S_2, there must be another segment $[p, p']$ on the cycle C, thus in the tree T, such that the points p and p' are in different sets of S_1 and S_2. Since $[p_1, p_2]$ is the shortest segment with two ends in different sets of S_1 and S_2, the segment $[p, p']$ is at least as long as the segment $[p_1, p_2]$. Replacing the segment $[p, p']$ in T by the segment $[p_1, p_2]$ gives a new Euclidean spanning tree T' of S such that the sum of the edge lengths of T' is not larger than the sum of the edge lengths of T. Since T is a Euclidean minimum spanning tree for S, the sum of the edge lengths of the tree T' must be the same as that of T. Therefore, T' is also a Euclidean minimum spanning tree for the set S and T' contains the segment $[p_1, p_2]$. $\square$

Corollary 7.4.3 *If a segment $[p_1p_2]$ is an edge of a Euclidean minimum spanning tree for the point set S, then the segment $[p_1p_2]$ is an edge in the Delaunay triangulation $D(S)$ of the set S.*

PROOF. The corollary follows from Lemmas 7.4.1 and 7.4.2 directly. $\square$

By Corollary 7.4.3, the Delaunay triangulation $D(S)$ of the point set S contains *all* segments that are in (any) Euclidean minimum spanning trees of the set S. Thus, if we take $D(S)$ as a weighted graph $G_{D(S)}$ in which the weight of an edge $[p_1, p_2]$ in $G_{D(S)}$, where $[p_1, p_2]$ is a segment in $D(S)$, is equal to the Euclidean distance between the points p_1 and p_2, then a Euclidean minimum spanning tree of the set S is a minimum weighted spanning tree of the graph $G_{D(S)}$. This suggests the following algorithm.

```
Algorithm EMST(S)
Input: a set S of n points in the plane
Output: a Euclidean minimum spanning tree of S
1. construct the Delaunay triangulation D(S) for the set S;
2. construct the weighted graph G_D(S);
3. apply Kruskal's algorithm on G_D(S) to find a MST T for G_D(S);
4. return(T).
```

The analysis of the algorithm `EMST` is simple. By Corollary 7.3.4, step 1 of the algorithm for constructing the Delaunay triangulation `D(S)` takes time $O(n \log n)$. To construct the graph `G_D(S)`, we simply compute the length of each edge in the Delaunay triangulation `D(S)`. Since `D(S)` is a planar graph of n vertices, the number of edges of `G_D(S)` is bounded by $O(n)$ (see section 2.4). Thus, step 2 of the algorithm takes in time $O(n)$. Kruskal's algorithm runs in time $O(m \log n)$ on a weighted graph with n vertices and m edges. Since the graph `G_D(S)` has $O(n)$ edges, the application of Kruskal's algorithm on `G_D(S)` takes time $O(n \log n)$. This gives the following theorem.

Theorem 7.4.4 *Given a set S of n points in the plane, the Euclidean minimum spanning tree of S can be constructed in time $O(n \log n)$.*

For completeness, we give a brief description of Kruskal's algorithm for constructing a minimum weighted spanning tree in a weighted graph. Since the algorithm has been well studied and given in many textbooks, the description here omits some details. The interested reader is referred to [2].

Kruskal's algorithm finds the minimum weighted spanning tree for a weighted graph G by simply adding edges one at a time, at each step using the lightest edge that does not form a cycle. The algorithm gradually builds up the tree one edge at a time from disconnected components. The correctness of the algorithm follows from a theorem for weighted graphs that is similar to our Lemma 7.4.2 for the Euclidean case.

To implement Kruskal's algorithm, suppose that the number of vertices of the graph G is n, and that the number of edges of G is m. We first presort all edges of G by their weights in non-decreasing order, then try to add the edges in order. The presorting of edges of G takes time $O(m \log m) = O(m \log n)$. We then maintain a forest F, which is a list of disjoint subtrees in the graph G. Each subtree T in the forest F is represented by a Union-Find tree whose nodes are the vertices of the subtree T [1] (to distinguish the subtrees in the forest F, which are subtrees in the weighted graph G, from the Union-Find trees that represent the subtrees in F, we call the vertices of the subtrees in F *vertices*, while call the vertices of the Union-Find trees

[1]For detailed discussions on the Union-Find structures, see [2], Section 4.7.

nodes). Initially, the forest F is a list of n trivial trees, each consists of a single vertex of G. Pick the next edge $e = [v, u]$ from the sorted list of edges of G, and check if v and u are in the same subtree in the forest F. This can be done by two Find operations followed by checking if the roots of the two corresponding Union-Find trees are the same. If v and u are in the same subtree in the forest F, then adding the edge $e = [v, u]$ would result in a cycle in the forest F. So we should not add the edge e to the forest F. On the other hand, if v and u are in different subtrees in the forest F, then the edge e does not form a cycle in the forest F, so we should add the edge e to the forest F. This is equivalent to merging the two corresponding Union-Find trees that contain the vertices v and u in F, respectively. This can be done by a single Union operation. We keep adding edges until the forest F becomes a single tree, which can be proved to be a minimum weighted spanning tree of the weighted graph G. Since for each edge in the graph, at most three Union-Find operations are performed, to construct the final minimum weighted spanning tree, we need at most $3m$ Union-Find operations. This can be done in time $O(m\alpha(n))$, where $\alpha(n) = o(\log(n))$ (see [2], Section 4.7 for detailed discussion). This leads to the conclusion that the running time of Kurskal's algorithm is $O(m \log n) + O(m\alpha(n)) = O(m \log n)$.

7.5 Maximum empty circle

Given a set S of n points in the plane, we are interested in finding a largest circle that does not contain any point in S. The problem has applications in areas such as scheduling and location planning. However, without further constraints, the problem is not very well-defined – from a place that is far enough from the point set S, we can make an arbitrarily large empty circle. Thus, we enforce a further constraint that the center of the circle must be contained in the convex hull of the point set S. This motivates the Maximum-Empty-Circle problem, which is formally defined as follows:

> Maximum-Empty-Circle
> given a set S of n points in the plane, find a largest circle that has its center in the convex hull CH(S) of S and contains no points of the set S in its interior.

Such a circle will be called a *maximum empty circle* for the set S. A maximum empty circle for a point set S can be specified by its center and radius.

We first consider where the center of a maximum empty circle can be located.

Lemma 7.5.1 *The center of a maximum empty circle for a set S of points in the plane must be either a Voronoi vertex of Vor(S), or an intersection of a Voronoi edge of Vor(S) and a boundary edge of the convex hull CH(S).*

PROOF. Suppose that C is a maximum empty circle for the point set S such that C is centered at a point q.

Since C is a maximum empty circle, the boundary of the circle C must contain at least one point of the set S – otherwise, we can increase the radius of C (without moving the center q of C) to get a larger empty circle.

If the boundary of the circle C contains only one point p_i in the set S, then we can move the center q of the circle C away from the point p_i, increase the radius of C, and keep the circle empty. This contradicts our assumption that C is a maximum empty circle.

Consequently, the center q of the circle C cannot be in the interior of any Voronoi polygon V_i of a point p_i in the set S – otherwise, the point p_i would be the only closest point in S to the center q and the boundary of the circle C cannot contain any other points in the set S except p.

Therefore, the point q must be on a Voronoi edge of the Voronoi diagram Vor(S) and the boundary of the circle C contains at least two points in the set S. Suppose that the point q is not on a Voronoi vertex of Vor(S), then there are exactly two points p_i and p_i in the set S on the boundary of the circle C. If q is not on the boundary of the convex hull CH(S), then we can move q along the Voronoi edge, which is the perpendicular bisector of p_i and p_j, in two opposite directions, in which one would move q away from both p_i and p_j (without getting out of the convex hull *CH(S)*), and increase the radius of the circle, thus getting a larger empty circle. This again contradicts the assumption that C is a maximum empty circle for the set S.

In conclusion, the center q of the maximum empty circle C must be either a Voronoi vertex in the Voronoi diagram Vor(S), or an intersection of a Voronoi edge in Vor(S) and the boundary of the convex hull CH(S). □

Let q be either a Voronoi vertex of the Voronoi diagram Vor(S) or an intersection of a Voronoi edge in Vor(S) and an edge of the convex hull CH(S). The radius of the largest empty circle centered at q can be computed easily. In fact, if q is a Voronoi vertex of Vor(S), then by Lemma 5.2.3, the point q is equidistant from three points in the set S and no points of S is in the interior of the circle defined by these three points. Therefore, the circle defined by these three points must be the largest empty circle centered at q. On the other hand, if q is an intersection of a Voronoi edge and a convex hull edge, then exactly two points p_i and p_j in the set S are closest to q, so the largest empty circle centered at q must have radius $|qp_i| = |qp_j|$.

If the Voronoi diagram Vor(S) is given by a DCEL, then in constant time, we can compute the radius of the largest empty circle centered at a Voronoi vertex v, by an algorithm `Trace-Vertex` that is similar to the algorithm `Trace-Region` in section 2.4, to traverse all incident Voronoi edges and all incident Voronoi polygons of the vertex v. (Note that a Voronoi vertex has degree exactly 3.) By Lemma 5.2.6, the Voronoi diagram Vor(S) has $O(n)$ Voronoi vertices. Thus, in linear time we can construct all largest empty circles that are centered at the Voronoi vertices of Vor(S). Note that not all these circles are candidates of the maximum empty circle for the set S: those largest empty circles that are centered at a Voronoi vertex that is outside the convex hull CH(S) should be excluded. We will discuss later how to find these Voronoi vertices that are outside the convex hull CH(S).

Now let us discuss the points that are intersections of Voronoi edges and the convex hull edges. The first question is: how many such intersections can there be?

Lemma 7.5.2 *There are at most $O(n)$ points that are intersections of Voronoi edges and convex hull edges.*

Proof. Since the convex hull CH(S) is convex, a Voronoi edge in Vor(S), which is either a straight line segment or a straight semi-infinite ray, can intersect CH(S) at at most two points. Moreover, by Lemma 5.2.6, the Voronoi diagram Vor(S) has at most $O(n)$ Voronoi edges. □

The following observation is also important.

Lemma 7.5.3 *Each boundary edge of the convex hull CH(S) intersects at least one Voronoi edge of Vor(S).*

Proof. If a boundary edge $e = [p_i, p_j]$ of CH(S) does not intersect any Voronoi edges, then the entire segment $[p_i, p_j]$ is contained in a single Voronoi polygon of *Vor(S)*. But this is not possible, since the points on $[p_i, p_j]$ that are very close to the point p_i should be contained in the Voronoi polygon for the point p_i, while the points on $[p_i, p_j]$ that are very close to the point p_j should be contained in the Voronoi polygon for the point p_j. □

For simplicity, we will call the intersections of the Voronoi edges and the convex hull edges that are not a Voronoi vertex, the *intersecting points*. An intersecting point q' is the *successor* of another intersecting point q if the partial chain on the boundary of the convex hull CH(S) from q to q', in clockwise ordering, contains no other intersecting points.

Lemma 7.5.4 *If we traverse the boundary of a Voronoi polygon clockwise, starting from an intersecting point q and leaving the convex hull, then we must encounter at least another intersecting point. The first intersecting point after q we encounter must be the successor of q.*

PROOF. Let the Voronoi polygon we are going to traverse be V_i. Since the point q is on the boundary of V_i and is an intersecting point, the Voronoi polygon V_i must have at least one vertex inside the convex hull CH(S) and at least one vertex outside the convex hull CH(S). Since we are traversing the boundary of V_i and leaving the convex hull CH(S), we must eventually come back and enter the convex hull CH(S) in order to reach the vertices of V_i that are inside CH(S). Therefore, the boundary of the polygon V_i must intersect CH(S) at at least another point. Let q' be the first intersecting point after q we encounter in the traversing. Since both the partial chain of V_i between q and q', and the partial chain of CH(S) between q and q' make only right turns, and because both V_i and CH(S) are convex, the partial chain of CH(S) between q and q' must be entirely contained in the Voronoi polygon V_i. This implies that no intersecting points are between the points q and q' on the partial chain of CH(S). Therefore, the intersecting point q' is the successor of the intersecting point q. □

Now it is quite clear how we find all intersecting points. We start with an intersecting point q, traverse in clockwise order a Voronoi polygon V_i in the direction of leaving the convex hull CH(S). We will encounter another intersecting point q', which is the successor of the intersecting point q. At the point q', we reverse the traversing direction and start traversing, from the point q', a Voronoi polygon that is adjacent to V_i, again in clockwise order and in the direction of leaving the convex hull CH(S). We will then encounter the successor of the intersecting point q', etc.. We repeat this procedure until we come back to the first intersecting point q.

We summarize this in the following algorithm.

```
Algorithm Find-Intersections
Input: the Voronoi diagram Vor(S) and the convex hull CH(S)
Output: all intersecting points of Vor(S) and CH(S)
1. find an intersecting point q_0; Output(q_0);
2. traverse a Voronoi polygon to find the successor q' of q_0;
3. While (q' <> q_0)
     Output(q');  q = q';
     reverse the traversing direction to traverse the adjacent
     Voronoi polygon to find the successor q' of q.
```

We analyze the algorithm, assuming that the Voronoi diagram Vor(S) is given by a DCEL and the convex hull CH(S) is given by a circular doubly-linked list. To find the first intersecting point `q_0`, we pick any boundary edge e of the convex hull CH(S), scan the DCEL representing the Voronoi diagram Vor(S) edge by edge, and check which edge of Vor(S) intersects e. By Lemma 7.5.3, e intersects at least one Voronoi edge in Vor(S). Thus, in time $O(n)$, we will find a Voronoi edge that intersects e and obtain the first intersecting point `q_0`. So step 1 of the algorithm takes time $O(n)$.

Starting from an intersecting point q, we traverse the part of a Voronoi polygon that is outside the convex hull CH(S). By Lemma 7.5.4, we will encounter the successor of q. For this, we have to check, for each Voronoi edge e we are traversing, if e intersects the convex hull CH(S). This seems to need $\Omega(n)$-time to check all boundary edges of the convex hull CH(S) for each Voronoi edge e. Fortunately, since each boundary edge of CH(S) contains at least one intersecting point (Lemma 7.5.3), the successor of q must be either on the boundary edge e_q of CH(S) where the intersecting point q is located, or on the boundary edge of CH(S) that is next to the boundary edge e_q. Therefore, for each Voronoi edge e we are traversing, we only have to check two boundary edges on the convex hull CH(S). As a result, each Voronoi edge can be processed in constant time. Moreover, each Voronoi edge that is outside the convex hull CH(S) is traversed at most twice since each Voronoi edge is on the boundary of exactly two Voronoi polygons. Therefore, the total time spent on step 2 and step 3 in the algorithm `Find-Intersections` is bounded by the number of Voronoi edges that are outside the convex hull CH(S), which is in turn bounded by the number of Voronoi edges of the Voronoi diagram Vor(S), which is, by Lemma 5.2.6, bounded by $O(n)$.

Thererfore, the time complexity of the algorithm `Find-Intersections` that finds all intersecting points is bounded by $O(n)$.

Finally, we discuss how to determine if a Voronoi vertex v is inside or outside the convex hull CH(S). In the algorithm `Find-Intersections`, all Voronoi vertices we encounter during traversing are outside the convex hull CH(S). So we can simply mark them and not use them as potential candidates for the center of the maximum empty circle. The question is, can there be any Voronoi vertex that is outside the convex hull CH(S) but not encountered in the traversing of the algorithm `Find-Intersections`? The answer is *no*, as explained in the following paragraph.

Let v be a Voronoi vertex of the Voronoi diagram Vor(S) that is outside of the convex hull CH(S). Suppose that v be on the boundary of a Voronoi polygon V_i. The Voronoi polygon V_i cannot be completely outside the convex hull Vor(S), since otherwise the corresponding point p_i in the set S would be outside the convex hull Vor(S). Therefore, the Voronoi polygon V_i intersects CH(S) at at least two points. Let q and q' be two intersecting points of the Voronoi polygon V_i and the convex hull CH(S) such that the vertex v is contained in the partial chain on the boundary of V_i from q to q' in clockwise ordering, and that no other intersecting points are on this partial chain. Then the algorithm `Find-Intersections` will eventually encounter the intersecting point q and traverse this partial chain in V_i from q to q'. Now the vertex v must be encountered.

Summarizing the above discussions gives the following algorithm for solving the problem MAXIMUM-EMPTY-CIRCLE.

```
Algorithm Maximum-Empty-Circle
Input: a set S of n points in the plane
Output: a maximum empty circle for S
1. sonstruct the Voronoi diagram Vor(S) and the convex hull CH(S);
2. call the algorithm Find-Intersections to find all intersecting
     points of Vor(S) and CH(S);
   mark all Voronoi vertices that are outside the convex hull CH(S);
3. For (each intersecting point q found in step 2)
      compute the largest empty circle centered at q;
4. For (each unmarked Voronoi vertex v)
      compute the largest empty circle centered at v;
5. Return the largest empty circle among those by steps 3-4.
```

Step 1 of the algorithm `Maximum-Empty-Circle` takes time $O(n \log n)$, by Theorem 5.3.7 and by, say, `Graham-Scan` algorithm. Step 2 of the algorithm takes time $O(n)$, as we discussed above. The other steps in the algorithm trivially take time $O(n)$, by Lemma 5.2.6 and Lemma 7.5.2. As a conclusion, we obtain the following theorem.

Theorem 7.5.5 *The problem* MAXIMUM-EMPTY-CIRCLE *can be solved in time $O(n \log n)$.*