ELSEVIER

Computational Geometry 4 (1994) 119–136

Computational Geometry
Theory and Applications

Computing the smallest k-enclosing circle and related problems*

Alon Efrat[a], Micha Sharir[a,*], Alon Ziv[b]

[a]*Department of Computer Science, Tel Aviv University, Ramat Aviv, 69978 Tel Aviv, Israel*
[b]*Department of Computer Science, The Technion, 32000 Haifa, Israel*

Communicated by Emo Welzl; submitted 16 February 1993; accepted 17 December 1993

Abstract

We present an efficient algorithm for solving the "smallest k-enclosing circle" (kSC) problem: Given a set of n points in the plane and an integer $k \leqslant n$, find the smallest disk containing k of the points. We present two solutions. When using $O(nk)$ storage, the problem can be solved in time $O(nk \log^2 n)$. When only $O(n \log n)$ storage is allowed, the running time is $O(nk \log^2 n \log n/k)$. We also extend our technique to obtain efficient solutions of several related problems (with similar time and storage bounds). These related problems include: finding the smallest homothetic copy of a given convex polygon P which contains k points from a given planar set, and finding the smallest disk intersecting k segments from a given planar set of non-intersecting segments.

Key words: Geometric optimization; Smallest enclosing circle

1. Introduction

Much attention has recently been given to problems of the following form: Given a set S of n objects and a parameter $k \leqslant n$, find a k-subset (namely, a subset of cardinality k) of the objects that optimizes some cost function, among all possible k-subsets.

This problem was studied for a variety of cost functions. Aggarwal et al. [3] solve this problem when the parameter to be optimized is the diameter of the k-subset (in

* Work on this paper by the second author has been supported by NSF Grant CCR-91-22103, and by grants from the U.S.-Israeli Binational Science Foundation, the G.I.F., the German-Israeli Foundation for Scientific Research and Development, and the Fund for Basic Research administered by the Israeli Academy of Sciences.

* Corresponding author.

SSDI 0925-7721(94)00003-E

time $O(k^{2.5} n \log k + n \log n)$), the variance of the k-subset (in time $O(k^2 n + n \log n)$), the size of an axis-parallel enclosing square, or the perimeter of an axis-parallel enclosing rectangle (both in time $O(nk^2 \log^2 n)$; the solution to the first problem has recently been improved to $O(n \log n)$ by Chew and Kedem [6].

In [11], Eppstein finds the minimum area k-gon whose vertices are points of S (in time $O(n^2 \log n + 2^{6k} n^2)$, and the minimum area convex polygon containing k points of S (in time $O(k^3 n^2 + n^2 \log n)$).

In this paper we give a fast algorithm for the (apparently simpler) problem of finding a k-subset with the smallest enclosing circle, stated formally as follows.

The "smallest k-enclosing circle" (kSC) problem. Given a set S of n points in the plane and an integer $k \leqslant n$, find the smallest disk containing k points of S.

See Fig. 1 for an illustration of the problem.

The problem arises in several applications. For example, suppose we want to partition the given set S into *clusters* of k points each. We can find the smallest k-enclosing circle of S, remove from S the k-subset that lies within the circle (this will be the first cluster), and repeat the procedure for the remaining set. This is a reasonable heuristic for clustering; it has the advantage that we can stop the process as soon as the smallest k-enclosing circle becomes too large, concluding that the remaining set can no longer be decomposed into 'sufficiently-dense' clusters of size k. We also note that the case $k = n$ gives us the well-known problem of finding the smallest enclosing disk of S, which can be solved in linear time using Megiddo's technique [20].

The smallest k-enclosing circle problem has recently been studied independently, by several other researchers using different techniques. Eppstein and Erickson [12] solve this problem in $O(nk \log k + n \log n)$ time with $O(n \log n + nk + k^2 \log k)$ space, and Datta et al. [8] give an algorithm with the same running time and with space improved to $O(n + k^2 \log k)$. After the first appearance of our paper, Matoušek gave a simple randomized algorithm [19] with expected running time $O(n \log n + nk)$ using $O(nk)$ space, or expected time $O(n \log n + nk \log k)$ with $O(n)$ space.

We present two algorithms for solving the problem, which differ in the amount of storage that they are allowed. When only $O(n \log n)$ storage is available, our second

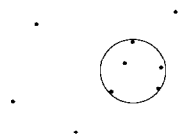

Fig. 1. A set of points S, and the smallest 5-enclosing circle of S.

algorithm solves the kSC problem in time $O(nk \log^2 n \log n/k)$. When we allow $O(nk)$ storage, we obtain, in out first (and simpler) algorithm, a slightly improved running time of $O(nk \log^2 n)$.

We can also apply our technique to obtain efficient solutions for other problems of this kind. Specifically:

- Given a set S of n points in the plane, a parameter $k \leqslant n$ and a convex polygon P with a constant number of edges, we can compute the smallest homothetic copy of P that contains k points of S, in the same running time and storage as in the two algorithms mentioned above.
- Given a set S of n non-intersecting segments in the plane and a parameter $k \leqslant n$, we can find a smallest disk intersecting k segments of S in the same time and storage as above.

Our algorithms make use of the parametric search technique, introduced by Megiddo in [21]. We will briefly describe the technique below, but we assume some familiarity of the reader with it; see the paper by Agarwal et al. [2] for an explanation of the technique and for other geometric applications of it.

The kSC problem for a planar set S is strongly related to higher-order Voronoi diagrams of S. Indeed, the smallest k-enclosing circle C passes either through three points of S or through two diametrically-opposite points. In the former case, assuming general position of the points in S, C contains $k-3$ points of S in its interior, and so its center is a vertex of the $(k-1)$-st order Voronoi diagrams of S, as is easily checked. In the latter case, the center of C lies on an edge of that diagram. Hence it suffices to compute the $(k-1)$-st order Voronoi diagram of S, go over all its edges and vertices, and find the one that minimizes its k-th smallest distance to the points of S, which can be easily computed in constant time per vertex, from the information available in the diagram. It follows that the smallest k-enclosing circle can be computed in time that is proportional to the time needed to compute the $(k-1)$-st order Voronoi diagram of S. The technique presented in this paper has several advantages over the higher-order Voronoi diagram technique: (i) it is simpler than the best known algorithm of Agarwal and Matoušek [1] for computing higher order Voronoi diagrams; (ii) it admits a storage-efficient solution; (iii) it can be easily extended to solve the related problems listed above; and (iv) it is somewhat more efficient — the algorithm of Agarwal and Matoušek [1] runs in time $O(n^{1+\varepsilon}k)$, for any $\varepsilon > 0$.

The paper is organized as follows: In Section 2 we present a simple $O(n^2 \log^2 n)$-time algorithm for solving the kSC problem; this algorithm is improved in Section 3 to get the two more efficient algorithms mentioned above. Section 4 extends these results to the computation of smallest (homothetic) k-enclosing polygons and of the smallest disk intersecting k of n given segments. Finally, Section 5 presents some open problems which are related to the ones solved in the paper, and gives a lower bound argument that suggests that the worst case complexity of the kSC problem is $\Omega(nk)$.

2. A naive approach

In this section we present a simple algorithm that solves the kSC problem in $O(n^2 \log^2 n)$ time and $O(n)$ space. The time bound will be improved later (at the cost of some increase in the storage requirement).

We first introduce some terminology. We assume that we are given a planar set $S = \{p_1, \ldots, p_n\}$ of n points. We also assume that the points of S are in *general position*, meaning in particular that no four points of S are co-circular. This assumption is not essential, but is made to simplify the description of our algorithms. Given a point p and a radius $r > 0$, we denote the circle of radius r centered at p by $C_r(p)$, and the closed disk bounded by that circle by $B_r(p)$. For any real number $r > 0$, we denote the arrangement of circles $C_r(p)$, $p \in S$, by $\mathcal{A}(S, r)$ (see Fig. 2). Given an arrangement of circles $\mathcal{A}$ and a point x, the *depth* of x in $\mathcal{A}$, denoted by $\text{depth}_{\mathcal{A}}(x)$, is the number of circles in $\mathcal{A}$ containing x (within the closed disks that they bound). We also define $\text{depth}(\mathcal{A}) = \max\{\text{depth}_{\mathcal{A}}(x) \mid x \in \mathbb{R}^2\}$.

The following two claims are obvious.

Claim 2.1. *Given a point set S, a point x and a positive number r,*

$$|S \cap B_r(x)| = \text{depth}_{\mathcal{A}(S,r)}(x).$$

Claim 2.2. *Given a point set S and a positive number r,*

$$\max\{|S \cap B_r(x)| : x \in \mathbb{R}^2\} = \text{depth}(\mathcal{A}(S, r)).$$

Based on these claims, we can rephrase our goal as "find the smallest value of r for which $\text{depth}(\mathcal{A}(S, r)) = k$". Let r^* denote this smallest radius. (As will follow from the analysis given below, our algorithms will also be able to produce a point having depth k in $\mathcal{A}(S, r^*)$, thus solving the original kSC problem.) This observation leads to a rather straightforward algorithm (described below), based on the parametric searching paradigm, for computing r^*. The first step in the design of such an algorithm is to produce an 'oracle' for determining whether any given radius r is too big or too small, as compared to r^*.

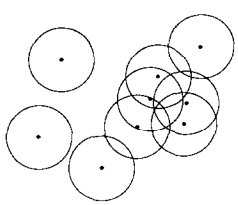

Fig. 2. The arrangement $\mathcal{A}(S, r)$.

2.1. Oracle for the naive approach

Our algorithm, which is based on the parametric search technique of Megiddo [21], performs an implicit binary search on certain 'critical' values of r to find r^* among them. In order to do this, the algorithm uses an 'oracle' procedure which is able to answer queries of the form "given a value r, is r^* less than, equal to, or greater than r?"

As we have observed, what we need to do is to determine whether depth($\mathcal{A}(S, r)$) is less than, equal to, or greater than k. In the first case, it is clear that $r < r^*$; similarly, in the third case we have $r > r^*$ (this follows from our general position assumption). For the second case (when the depth is exactly k), we only know that $r \geq r^*$. To differentiate between these two possibilities, we need to determine whether depth($\mathcal{A}(S, r)$) becomes strictly smaller than k when r is decreased by an arbitrarily small amount. This can happen only if two circles in $\mathcal{A}(S, r)$ are tangent to each other or three of these circles are concurrent, and if the only points at depth k are such points of tangency or of triple intersection. By examining the edges and vertices of $\mathcal{A}(S, r)$, we can easily tell whether this situation arises, thus resolving the comparison between r and r^*.

Our oracle is based on the observation that if there is a point of depth k in $\mathcal{A}(S, r)$, then there must exist a point of depth $\geq k$ on one of the circles; moreover, if at least two of the circles in $\mathcal{A}(S, r)$ intersect, one of these intersection points must also be of depth $\geq k$ (all points on a circle that does not intersect any other circle are at depth 1, while any intersection point of two circles is at depth at least 2). Hence it suffices to construct the vertices of the arrangement $\mathcal{A}(S, r)$, and compute the maximum depth of a vertex.

In order to reduce the amount of storage required by the algorithm, and also for the purpose of obtaining a parallelizable algorithm, which is required by the parametric searching technique, we implement the oracle by applying the following procedure to each of the circles $C_r(p_i)$, for $i = 1, \ldots, n$. The procedure computes the maximum depth of points that lie on its input circle; the maximum of all these output depths is the required depth($\mathcal{A}(S, r)$).

Procedure Calc-Depth(r, i)

1. Find the circles $C_r(p_j)$ that intersect $C_r(p_i)$. If no such circle exists, return 1 and exit.
2. Find the depth of the leftmost point of $C_r(p_i)$.
3. Sort the intersection points between $C_r(p_i)$ and the other circles by their (clockwise) order along $C_r(p_i)$, starting at the leftmost point of that circle.
4. Scan the sorted list of intersection points, and compute the depth of each of these points, observing that the depth of a point along $C_r(p_i)$ changes by ± 1 as we cross any point in the list, depending on whether we enter or exit a disk at that point.
5. Return the maximum depth computed at the preceding step.

It is easy to see that the time required by a call to *Calc-Depth* is $O(n \log n)$, so the total time required by a call to our oracle is $O(n^2 \log n)$. Since each circle is processed

separately, the storage required is $O(n)$. However, in what follows we will use the oracle in a different, more fragmented manner, as described in the following subsection.

2.2. The generic algorithm

Fix an index $i = 1, \ldots, n$. Denote the smallest radius r for which there exists a point on $C_r(p_i)$ with depth k by r_i^* (note that r_i^* is well defined for any k). Following the parametric searching paradigm, we attempt, for each $i = 1, \ldots, n$, to perform Calc-Depth(r_i^*, i) in a generic manner, without knowing the value of r_i^*. For this, we need to resolve the comparisons that depend on r_i^* which are executed by the procedure. Each of these resolutions will further restrict the range in which r_i^* can lie, so that at the end of the generic execution, the control flow of the procedure is the same for all values of r in this range, which easily implies that r_i^* is the smallest point in the final range. We apply this procedure to all indices $i = 1, \ldots, n$ and compute $r^* = \min_i r_i^*$.

In more detail, this is done as follows. The first step of Calc-Depth(r_i^*, i) is to find the circles that intersect $C_{r_i^*}(p_i)$. For any pair of points, p_i, p_j, the circles $C_r(p_i)$ and $C_r(p_j)$ intersect if and only if $r \geqslant r_{ij} \equiv \frac{1}{2}|p_i - p_j|$. We thus collect the $n - 1$ values r_{ij}, for $j \neq i$, and run a binary search among them to locate r_i^*, by $O(\log n)$ oracle calls of the form Calc-Depth(r_{ij}, i). This search determines the relative order between r_i^* and each of the r_{ij}'s, so each of the comparisons can now be resolved. (Note that here we are not using the full power of the oracle, since we want to compare the r_{ij}'s with r_i^* and not with r^*; thus it suffices to resolve these comparisons with calls to Calc-Depth involving only p_i, which costs us only $O(n \log n)$ time.)

The second step counts how many other circles contain the leftmost point of $C_{r_i^*}(p_i)$ within their disk. Again, given p_i and p_j, the leftmost point of $C_r(p_i)$ is contained in $B_r(p_j)$ if and only if p_j lies to the left of p_i and $r \geqslant r'_{ij}$, where r'_{ij} is the distance from p_i to the intersection point between the perpendicular bisector to $p_i p_j$ and the horizontal line through p_i. We thus compute all these critical values and run a binary search to locate r_i^* among them, using $O(\log n)$ calls to Calc-Depth, as above. This allows us to compute the depth of the leftmost point of $C_{r_i^*}(p_i)$.

The third step of Calc-Depth sorts the intersection points of $C_{r_i^*}(p_i)$ with the other circles of that radius. Here we use (a serial simulation of) a parallel sorting scheme that uses $O(n)$ processors and $O(\log n)$ parallel steps, such as the scheme of Ajtai et al. [4]. Each parallel step of the sorting attempts to perform $O(n)$ comparisons, each asking for the relative order along $C_{r_i^*}(p_i)$ of two of its intersections with, say $C_{r_i^*}(p_j)$ and $C_{r_i^*}(p_k)$. It is easy to see that, as we vary the radius of these circles (in the range where all 4 intersections between those pairs of circles exist), their relative order along $C_r(p_i)$ can change only when r is the circumradius r_{ijk} of the triangle $p_i p_j p_k$. Hence we obtain, for each parallel step of the algorithm, $O(n)$ critical values r_{ijk} for the radius, and we run a binary search among them as above, locating r_i^* among them and resolving all comparisons of that step. This allows us to complete the generic sorting step of Calc-Depth.

Finally, the fourth stage of Calc-Depth can be performed in a non-generic manner, since the information obtained in the three previous steps fully determines the maximum depth along $C_{r_i^*}(p_i)$. For this reason, we do not perform this step at all, since it provides no extra restriction on the allowed range of r_i^*.

The binary searches in the first three stages of the generic execution progressively narrow the range where r_i^* is known to lie. As noted above, the minimum value of this range upon termination of the generic execution is the desired r_i^*.

Concerning the running time of this generic simulation, the first two stages require time $O(n \log^2 n)$ (this time is dominated by $O(\log n)$ calls to Calc-Depth with specific radii). The third step takes $O(n \log^3 n)$ time, but using a standard trick due to Cole [7], this can be reduced to $O(n \log^2 n)$. Repeating this procedure for each p_i, we obtain r^* as the minimum of all the r_i^*'s, at a total cost of $O(n^2 \log^2 n)$. We will improve this naive bound in subsequent sections of the paper.

3. The improved algorithms

The main bottleneck of the previous algorithm is that it has to check for intersections between every pair of circles. To avoid this, we first claim that there exists some initial radius r_{init}, for which $depth(\mathscr{A}(S, r_{\text{init}}))$ is at least k, but the number of intersections between the circles in this arrangement is only $O(nk)$. The existence of such a radius is a consequence of the following lemma; a generalized version of the lemma is stated in Lemma 4.1 of the next section.

Lemma 3.1 (Pach; see [24]). *If an arrangement $\mathscr{A}$ of n circles in the plane contains at least $9nk$ intersecting pairs of circles, then* $\text{depth}(\mathscr{A}) \geq k$.

(Note that the lemma does not require the circles to be congruent, so it is stronger than what we need here.) The lemma allows us to replace, temporarily, the original problem with the following, simpler problem: "Given S and k as above, find the smallest radius r_{init} for which the number of pairs of intersecting circles in $\mathscr{A}(S, r_{\text{init}})$ is at least $9nk$". (By assuming appropriate general position, the number of intersecting pairs of circles in $\mathscr{A}(S, r_{\text{init}})$ is exactly $9nk$.) Lemma 3.1 implies that $r^* \leq r_{\text{init}}$. Our solution to this problem will also yield, for each circle $C_{r_{\text{init}}}(p)$, a list $L(p)$ of all circles intersecting $C_{r_{\text{init}}}(p)$ in $\mathscr{A}(S, r_{\text{init}})$; the total length of these lists is $O(nk)$.

After computing r_{init}, we will execute a variant of the algorithm described in the previous section, with the following two modifications: (i) we allow oracle calls only for $r \leq r_{\text{init}}$; and (ii) we use the lists $L(p)$ to find all vertices of $\mathscr{A}(S, r)$ along each of the circles $C_r(p)$ (this is done both in the generic algorithm and in the oracle calls). This is easily seen to reduce the running time of the previous algorithm to $O(nk \log^2 n)$, excluding the initial stage that computes r_{init}.

We now describe in detail this initial stage. It is also based on the parametric searching technique. As above, we first describe the oracle used in the binary searches and then describe the generic simulation at the unknown radius r_{init}.

Before going into the analysis, we remark that r_{init} can be computed in a straightforward manner using the recent algorithm of Lenhof and Smid [15], which produces the t closest pairs in a given planar set of n points, in (optimal) time $O(t + n \log n)$. If we apply this algorithm with $t = 9nk$, and let r_{init} be half the t-th smallest distance, we obtain the required r_{init}, in time $O(n(k + \log n))$. However, we develop here an alternative technique because: (i) it can be tuned so that it uses only $O(n \log n)$ storage; (ii) it can be easily generalized to shapes other than circles, as is discussed in the next section. Also, the saving in time in this stage does not change the overall asymptotic running time of the algorithm, which is dominated by the $O(nk \log^2 n)$ cost of the subsequent phase.

3.1. The oracle: a simple version

The oracle has to answer queries of the form: "Given S, k and r as above, determine whether the number of pairs of intersecting circles in $\mathcal{A}(S, r)$ is less than, or equal to, or larger than $9nk$". Depending on the answer to the query, we have respectively $r < r_{\text{init}}$, $r = r_{\text{init}}$, and $r > r_{\text{init}}$ (for this we need to assume, as noted above, an appropriate version of general position). The oracle is performed using a straightforward line-sweep algorithm that counts the number of intersections between the circles. If we execute the oracle for a value of r that is greater than r_{init}, the number of intersections can be too big, and we therefore stop the oracle as soon as it encounters more than $9nk$ pairs of intersecting circles, concluding that $r > r_{\text{init}}$ in this case. Hence the running time of the oracle is $O(nk \log n)$, and it uses $O(n)$ storage.

3.2. The oracle: an improved parallelizable version

We next describe another algorithm that determines whether the number of pairs of intersecting circles in the arrangement is less than, equal to, or larger than $9nk$, which is easier to parallelize. The algorithm uses only $O(nk)$ processors, and runs in $O(\log^c n)$ parallel steps (where the constant c depends on the storage we have at hand).

The algorithm requires a data structure $\text{DS}(X)$ for answering efficiently queries of the following form, for a given planar set of points X: "Given a query point p and a parameter r, determine whether $X \cap B_r(p)$ is empty." To this end we use the Voronoi diagram of X, augmented with an efficient point location structure. To find whether $X \cap B_r(p) = \emptyset$, we simply find the point $q \in X$ nearest to p, and check whether the distance $d(p, q)$ is larger than r.

We build a fully balanced binary tree $\mathcal{T}$. The leaves of $\mathcal{T}$ represent the points of S, and each node v of $\mathcal{T}$ represents the subset S_v of S consisting of all points stored at the leaves of the subtree rooted at v. With each node v of $\mathcal{T}$ we also store $\text{DS}(S_v)$.

We note that $\mathcal{T}$ can be constructed in time $O(n \log n)$, as follows. We first sort the points of S by their x-coordinates, and put them in the leaves of $\mathcal{T}$ in this order. We then construct $\mathcal{T}$ bottom-up; at each new node v we need to merge $DS(S_{v_1})$ and $DS(S_{v_2})$, where v_1 and v_2 are the children of v. This amounts to merging the Voronoi diagrams of the two sets S_{v_1} and S_{v_2} to obtain the diagram of S_v. Since these two subsets are separated, the two diagrams can be merged in linear time, using e.g. the Shamos–Hoey technique (see [22]). In addition, we need to further process the Voronoi diagram of S_v for efficient point location. This can also be performed in linear time, using the technique of Edelsbrunner et al. [9] or of Seidel [23]. Hence the total time needed for constructing $\mathcal{T}$ is $O(n \log n)$. This cost will be dominated by the cost of the subsequent phases of the algorithm. Note also that $\mathcal{T}$ requires $\Theta(n \log n)$ storage. Our goal now is to find, for each $p \in S$, the set of points q such that $C_r(p)$ intersects $C_r(q)$ (or rather to count the size of this set). This is exactly the set of points for which $d(p, q) \leqslant 2r$. We want to add up the sizes of these sets, and stop the oracle as soon as the sum first exceeds $9nk$. The algorithm uses a total of (at most) $18nk + n$ processors. Initially, it allocates a single processor to each point $p_i \in S$, and starts the procedure Count (described below) with the root of $\mathcal{T}$ and the point p_i as parameters. The calls to Count are synchronized so that each level of the tree is processed simultaneously by all processors. (This last condition is made for clarity of exposition, and can be relaxed — see below.)

The procedure Count gets a node v of $\mathcal{T}$ and a point $p \in S$ as parameters; it is also allocated a processor which executes it. The procedure runs as follows.

Procedure Count(v, p)

1. If v is a leaf, check whether the point p_v stored at v is different from p, and if its distance from p is $\leqslant 2r$. Add in this case 1 to a global count N.
2. Otherwise, let v_1 and v_2 be the two children of v. Use $DS(S_{v_1})$ and $DS(S_{v_2})$ to determine whether $S_{v_1} \cap B_{2r}(p)$ and $S_{v_2} \cap B_{2r}(p)$ are nonempty. Call *Count* recursively with v_1 (resp. v_2) and p, if the first (resp. second) intersection is nonempty. Allocate a new processor if both calls are needed (so that one call uses the old processor allocated at v and the other call uses the new processor).

It can be readily verified that the value N reported by the algorithm (as accumulated from all the calls to Count) is twice the number of pairs of intersecting circles in $\mathcal{A}(S, r)$. Also, this number is at least as large as the number of processors at each level (except for the root) minus n; indeed, we perform a recursive call with a point p at a node v only if we are guaranteed that $B_{2r}(p)$ contains at least one point q of S_v. If $p \neq q$, this pair of points will generate two processors at each level, and will eventually cause N to be increased by 2. If $p = q$, this pair generates a single processor at each level, without affecting N, for a total of n 'spurious' processors per level. Therefore, if at the beginning of the processing of a level the total number of processors allocated is $> 18nk + n$, the algorithm can stop immediately and report that the number of pairs

of intersecting circles in $\mathscr{A}(S, r)$ is larger than $9nk$. (The technique used here is well known; see for example [5].)

3.3. The generic algorithm

We next describe the generic algorithm that simulates the procedure Count at the unknown value r_{init}. We first describe a version that is allowed to use $O(nk)$ storage, and then explain how to modify it to reduce the storage to $O(n \log n)$, at the cost of a slight increase in the running time.

The procedure processes each of the $O(\log n)$ levels of $\mathscr{T}$ separately. At an intermediate level, each of the $O(nk)$ processors locates a point p_i in the Voronoi diagram of some subset S_v of S. These locations are independent of r_{init}, so they can be performed explicitly, in $O(\log n)$ time per operation. Then (also at the leaf level) the distance from p_i to its nearest neighbor in S_v is compared with $2r$. We thus need to run a binary search to locate r_{init} among these $O(nk)$ distances, using $O(\log nk) = O(\log n)$ calls to the simple sweep-based version of the oracle, each of which takes time $O(nk \log n)$. Thus the total cost of the generic simulation is $O(nk \log^3 n)$. (Note that, since the generic simulation follows the execution of the oracle at r_{init}, we are guaranteed that the generic execution does not require more than $18nk + n$ processors.)

This can be improved to $O(nk \log^2 n)$ using Cole's trick [7]. This amounts to executing only a constant number of binary search steps at each stage of the algorithm, thereby resolving only some fixed large fraction of the number of comparisons. Nodes whose comparisons were resolved can proceed to the next level while the other nodes are stuck and have to participate again in the next round of binary search. It is easily verified that Cole's technique is indeed applicable here. In particular, one needs to observe that, at any stage of the revised algorithm, where processors can now reside at different levels of the tree, it is still true that if the number of processors exceeds $18nk + n$ then the number of pairs of intersecting circles in $\mathscr{A}(S, r)$ exceeds $9nk$. Hence, arguing as above, the generic execution does not require more than $18nk + n$ processors.

As above, at the end of the generic simulation we obtain an interval where r_{init} can lie, and the smallest endpoint of that interval is r_{init}.

In summary, including the $O(n \log n)$ time used by the initial stage that constructs $\mathscr{T}$, and the $O(nk \log^2 n)$ time used by the subsequent phase that computes r^*, we obtain a first version of the improved algorithm, which requires $O(nk \log^2 n)$ time and $O(nk)$ storage.

3.4. A space-efficient version

In the version just presented, we have to know explicitly which pairs (p, v) of a point $p \in S$ and a node v of $\mathscr{T}$ are active at any given time, and therefore $\Theta(nk)$ storage is

required. The storage requirement can be reduced to $O(n \log n)$, at the cost of a slight increase in running time, as follows. For each $p \in S$, denote by *live*(p) the set of nodes of $\mathcal{T}$ that the algorithm currently searches with point p (at any execution instance). Since we can no longer maintain explicitly all the sets live(p), we will reconstruct each of these sets from scratch, whenever such a set is needed (which, fortunately, does not occur too often).

There are some technical difficulties in this approach. To see the issues that can arise, consider first the following initial attempt at the problem. When we process the j-th level of $\mathcal{T}$, we iterate over the points $p \in S$, and, for each point p, we reconstruct live(p) by executing Count with p, starting at the root of $\mathcal{T}$ and proceeding until level j is reached. We then collect all critical values of r that the comparisons at level j generate, and compute their median $r_m(p)$. We repeat this process for each $p \in S$, obtaining n median values $r_m(p_1), \ldots, r_m(p_n)$. We compute the median r_m of these n medians, and call the (simple sweep-based) oracle at r_m. This resolves half of the comparisons at live(p), for half of the points p of S. (We would, of course, prefer to maintain all the critical values of r for all points p, but this requires too much storage.)

The problem with this approach is that now we have to repeat this procedure, but only with the critical values of r that were not yet resolved. If we still do it one level at a time, we will be spending too much time to resolve all comparisons at each level. To make the technique more efficient, we note that the generic algorithm maintains at all times an interval I where r_{init} is known to lie. Comparisons whose critical values of r lie outside I can be resolved immediately, while comparisons having a critical value inside I can be resolved only by further calls to the oracle. Our revised strategy is thus to proceed, with each $p \in S$ in turn, down the tree $\mathcal{T}$ as deep as possible, until we encounter comparisons that cannot yet be resolved. We now denote by live(p) the resulting set of nodes where the comparisons involving p get 'stuck'. To make this method efficient, we use (an appropriate modification of) the weighing technique of Cole [7]. That is, we give each stuck comparison at level j a weight of 4^{-j} (so that comparisons stuck higher in the tree are more important since they need to be resolved more urgently). For each point p, and each stage of the algorithm, we compute the weighted median $r_w(p)$ of the critical values of r at the nodes of live(p); we give $r_w(p)$ a weight equal to the sum of the weights of the nodes of live(p). We then compute the weighted median r_w of these medians, and call the oracle with r_w, thus resolving comparisons whose total weight is 1/4 of the overall weight of all comparisons. It can be shown, arguing as in [7], that the total weight of stuck comparisons is reduced by a constant factor at each stage of the algorithm, so that, after at most $O(\log n)$ stages, we are guaranteed that no comparisons will be stuck, so the generic execution can be completed. Clearly, the storage used by the algorithm is dominated by the size of $\mathcal{T}$, and is thus $O(n \log n)$. (Note that, arguing as in the preceding version of the algorithm, we are guaranteed that the total size of all the sets live(p), at any given moment, does not exceed $O(nk)$.)

The running time of the procedure is estimated as follows. The total cost of all oracle calls, as just argued, is $O(nk \log^2 n)$. After each oracle call, we need to

re-calculate the sets live(p), for each $p \in S$. This cost is $\sum_{p \in S} O(|\mathcal{T}_p| \log n)$, where $\mathcal{T}_p$ is the subtree of $\mathcal{T}$ whose root is the root of $\mathcal{T}$ and whose leaves are the nodes of live(p). Thus the total overhead of these re-calculations is $O(\log^2 n \cdot \sum_{p \in S} |\mathcal{T}_p|)$. To obtain a bound on this sum, we note that if a subtree of $\mathcal{T}$ has d leaves then the number of its internal nodes is $O(d \log n/d)$, as is easily checked. Thus, if live(p_i) has d_i nodes, then the corresponding sum is $O(\sum_{i=1}^{n} d_i \log n/d_i)$, and we are also given that $\sum_{i=1}^{n} d_i \leqslant 18nk + n$. Simple calculation shows that the maximum value of the sum is $O(nk \log n/k)$. Thus the total overhead of the re-calculations of the sets live(p) is $O(nk \log^2 n \log n/k)$, which is easily seen to dominate the overall running time of this portion of the algorithm (including the time used by the initial construction of $\mathcal{T}$).

Recall that we have so far been discussing only the initial phase of the algorithm, which computes r_{init}. The rest of the algorithm continues by using the naive oracle and its generic simulation. However, every (generic or explicit) call to Calc-Depth with a specific point p requires availability of the corresponding list $L(p)$, and in this version of the algorithm we cannot afford to maintain all these lists simultaneously. We overcome this difficulty as follows: When we start a generic simulation of Calc-Depth with a specific $p \in S$, we first go down the tree $\mathcal{T}$ and compute the set live(p) at the value $r = r_{\text{init}}$; note that this set now consists only of leaves of $\mathcal{T}$ and thus gives the set of all circles $C_{r_{\text{init}}}(q)$ that intersect $C_{r_{\text{init}}}(p)$. We then proceed with the generic execution of Calc-Depth using only the circles given by live(p). The extra cost of this final calculation of the sets live(p), and the cost of the generic execution of the calls to Calc-Depth, are clearly dominated by the cost of the initial stage that calculates r_{init}.

We thus conclude the following.

Theorem 3.2. *Given a set S of n points in the plane and an integer $k \leqslant n$, one can compute the smallest circle containing k points of S in time $O(nk \log^2 n)$ and space $O(nk)$, or in time $O(nk \log^2 n \log n/k)$ and space $O(n \log n)$.*

4. Extending the algorithm to other shapes

4.1. Finding the smallest k-enclosing homothetic copy of a polygon

Let P be a given convex polygon with d sides, where we consider d to be a constant. We wish to extend the technique presented above to solve the following problem. "Given a set S of n points in the plane and a parameter $k \leqslant n$, find the smallest homothetic copy of P that contains k points of S."

We choose some arbitrary center point c inside P, and regard P as given in a fixed position in which c lies at the origin. We denote any homothetic copy of P as $\alpha P + v$, where $\alpha > 0$ is the *scaling factor* of P and v is the location of the center c of this copy. It is easily verified that there exists a homothetic copy $\alpha P + v$ of P which contains k points of S if and only if there exists a point that lies in k of the n polygons $\alpha P^- + p_i$,

for $p_i \in S$, where $P^- = (-P)$ is the reflection of P about c. We denote by $\mathscr{A}_P(S, \alpha)$ the arrangement formed by these polygons, and assume that S and P are in general position, as above. Then this condition is equivalent to the existence of a vertex of $\mathscr{A}_P(S, \alpha)$ at depth k (where the depth of a point x is defined, in complete analogy to the definition given above, as the number of polygons that contain x in their interior or on their boundary).

We can thus apply the same machinery developed in the preceding sections, with a few necessary modifications. That is, we need an oracle which, given α and k, determines whether there exists a vertex of $\mathscr{A}_P(S, \alpha)$ at depth k. We first note that, like circles, isothetic (and even homothetic) copies of a convex polygon have the property that, under an appropriate general position assumption, each pair of their boundaries intersect in at most 2 points (cf. Kedem et al. [14]). Thus the naive algorithm can proceed in much the same way as above; the cost of the primitive operations that it performs may now depend on d, but since we assume d to be a constant, they still take constant time each.

In order to extend the improved algorithm of Section 3, we need to extend Pach's lemma (Lemma 3.1) to shapes other than circles. This is indeed possible as follows.

Lemma 4.1 (Sharir [24]). *Let $\mathscr{A}$ be an arrangement formed by n shapes in the plane, having the property that the boundaries of each pair of shapes intersect at most twice. If the maximal depth of the arrangement is $\leqslant k$ then the number of intersection pairs of the boundaries is $\leqslant cnk$, for some absolute positive constant c.*

Remark 4.2. The lemma is stated in more generality than is actually needed here, since in our application all copies of P are isothetic.

Another modification that the new procedure requires is in the preliminary stage that computes r_{init} (or, in our new notation, α_{init}, the smallest scaling factor for which the arrangement $\mathscr{A}_P$ has at least cnk intersecting pairs of polygons (or, under an appropriate general position assumption, exactly cnk intersecting pairs)). The data structure that was used in Section 3 is based on standard Voronoi diagrams of subsets of S. Here we need to use the following generalized structure. The basic operation that we want our data structure to support is: Given a fixed subset $X \subseteq S$, a query point q and a query scaling factor α, determine whether there is $x \in X$ such that $(\alpha P^- + x) \cap (\alpha P^- + q) \neq \emptyset$. This is equivalent, as is easily checked, to asking whether $X \cap [q + \alpha(P - P)] \neq \emptyset$. Thus we can compute the generalized Voronoi diagram of X under the norm induced by $P - P$ as a unit ball (see [18] for details), and preprocess it for efficient point location. The above query is answered by locating q in the diagram, and by determining whether the $(P - P)$-distance from q to its nearest neighbor in X is $\leqslant \alpha$.

It is now easy to see that all the algorithms described in the previous sections can be applied for the case of a convex polygon with a constant number of sides, or, for that

matter, for any convex region P of sufficiently simple shape so that each of the basic operations performed by the algorithm on O(1) copies of P takes constant time.

Remark 4.3. If d is not assumed to be a constant, we can apply a simple prune-and-search technique (cf. Kirkpatrick and Snoeyink [16]), and an efficient Voronoi diagram for convex distance functions, such as described by Kao and Mount [17] and modified as suggested by Kirkpatrick and Snoeyink [16], to obtain an algorithm whose time complexity is worse than the above bounds by a factor of $O(\log d)$.

Remark 4.4. If the given polygon is a square, the problem can be solved more efficiently, in time $O(n \log n)$ (see Smid [25], and also Chew and Kedem [6], where an $O(n \log^2 n)$ algorithm is presented).

4.2. Finding the smallest disk intersecting k segments

We now describe how the algorithm can be modified, so as to find the smallest disk intersecting k segments out of a given set S of n non-intersecting segments in the plane.

We can solve this problem by the technique described in the previous section, with the following modifications: First, given a distance $r > 0$ we define the hippodrome $H(e, r)$ of a segment $e \in S$ to be the set of all points of distance $\leqslant r$ from e; See Fig. 3 for an illustration.

We note that a disk D of radius r intersects k of the segments of S if and only if the center of D lies in k of the hippodromes $H(e, r)$, for $e \in S$. To compare r with the radius r^* of the smallest disk we seek, it suffices to determine whether there exists a point whose depth in the arrangement $\mathcal{A}_r$ of the hippodromes $H(e, r)$ is $\geqslant k$ (the depth is define exactly as in Section 2). To apply the mechanism of the preceding sections, we use the following property, which is a consequence of the analysis of Kedem et al. [14].

Claim 4.5. *Let e_1, e_2 be two non-intersecting convex shapes in the plane, and let H_i, for $i = 1, 2$, be the r-neighborhood of e_i, that is, all points of distance $\leqslant r$ from e_i. Then, assuming general position of e_1 and e_2, the boundary of H_1 intersects the boundary of H_2 at most twice.*

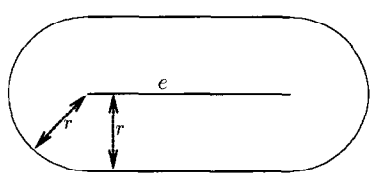

Fig. 3. The hippodrome $H(e, r)$.

This claim, combined with Lemma 4.1, implies that we can define r_{init} to be the smallest radius r for which the arrangement $\mathcal{A}_r$ has at least cnk vertices, and the desired optimal radius r^* will be $\leqslant r_{\text{init}}$. The naive algorithm can be extended to this case with only few straightforward modifications. The improved algorithm can also proceed as in the previous section, exploiting the above definition of r_{init}. The only modification that requires some comment is in the construction of the tree $\mathcal{T}$. Here we need to use the Euclidean Voronoi diagrams of subsets of S. We construct these diagrams using Yap's algorithm [25], which takes time $O(m \log m)$ for a subset of m segments. Here we do not know how to merge two subdiagrams in linear time, as was done in section 3. So we simply construct each Voronoi diagram in $\mathcal{T}$ from scratch, requiring a total of $O(n \log^2 n)$ time, as is easily seen, which is anyway subsumed by the running time of the remainder of the algorithm.

In summary, we have thus shown the following.

Theorem 4.6. *Given a set S of n points in the plane, an integer $k \leqslant n$, and a convex d-gon (where d is a constant), we can find the smallest homothetic copy of the polygon containing k points of S, in time $O(nk \log^2 n)$ and space $O(nk)$, or in time $O(nk \log^2 n \log n/k)$ and space $O(n \log n)$.*

Theorem 4.7. *Given a set S of n non-intersecting segments in the plane and an integer $k \leqslant n$, the smallest disk intersecting k of the segments of S can be found in time $O(nk \log^2 n)$ and space $O(nk)$, or in time $O(nk \log^2 n \log n/k)$ and space $O(n \log n)$.*

Additional extensions of this kind can easily be devised, for other shapes of the desired enclosing region, and also for sets S of objects other than points or segments. We leave it to the interested reader to further explore these extensions.

5. Conclusion

In this section we conclude the paper with some related open problems and with a lower bound argument for the kSC problem.

5.1. Open problems

The following problems are related to the problems solved in this paper, and seem to be still open.

Rotation. How fast can one find the translation *and rotation* of a given segment minimizing the k-th smallest distance to a given set of n points in the plane? (The special case $k = n$ has been solved by Efrat and Sharir [10] in near-linear time.

Minimal ring width. Find the ring of smallest width containing k of the given points. (The special case $k = n$ has been solved in Agarwal et al. [2] in time $O(n^{8/5+\varepsilon})$, for any $\varepsilon > 0$.)

General polygons. The restriction in Section 4 that the shapes considered there be convex seems somewhat artificial, although this property is crucial for our analysis. How fast can the problem be solved for more general figures, not necessarily convex?

5.2. *Lower bound*

All the known algorithms for solving the smallest k-enclosing circle problem, ours as well as those mentioned in the introduction of this paper, run in time $O(nk \operatorname{polylog} n)$. In this subsection we will provide a combinatorial analysis of the problem that suggests that these running times are close to being tight in the worst case. Specifically, we construct a set S of n points in the plane, for which there exists a set D of $\Omega(nk)$ disks, all having roughly the same radius, and each is the smallest disk enclosing some $2k$-subset of S. Assume now that each point p_i of S is perturbed by some small vector v_i, and that the disks of D are modified so that each disk is still the smallest disk enclosing the same subset it enclosed before the perturbation. Then it does not seem likely that one could find the smallest modified disk of D without inspecting all discs in D, which takes $\Omega(nk)$ time.

We first construct[1] a set D_1 of $4k$ disks of equal radius, so that the complexity of the region R of all points contained in exactly $2k + 2$ disks is $\Omega(k^2)$. For this, we pair up these disks so that the intersection of each pair forms a (sufficiently thin) slice. We place the $2k$ slices so that k of them are horizontal, k are vertical, and each horizontal slice intersects every vertical slice; See Fig. 4. Observe that this arrangement of disks has $\Omega(k^2)$ cells of depth $2k + 2$, where each such cell is the region of overlap between a horizontal slice and a vertical slice; each cell c is thus contained in a different subset D_c of $2k + 2$ disks, and hence the center of the smallest circle enclosing the centers of disks of D_c lies in c. Let S_1 denote the set of the centers of the disks of D_1. We have thus constructed $\Omega(k^2)$ nearly congruent disks, each being the smallest disk containing a subset of $2k + 2$ points of S_1.

To establish the $\Omega(nk)$ bound, we place $n/4k$ copies of D_1 in the plane, sufficiently far from each other, so that no two disks of different copies intersect. Let D denote the set of these n disks, and let S denote the set of their centers. Arguing as above, it is easily seen that the number of smallest $(2k + 2)$-enclosing circles for S is

$$\frac{n}{4k} \times \Omega(k^2) = \Omega(nk),$$

and that all these circles are nearly congruent.

[1] The authors wish to thank Pankaj Agarwal for bringing this construction to our attention.

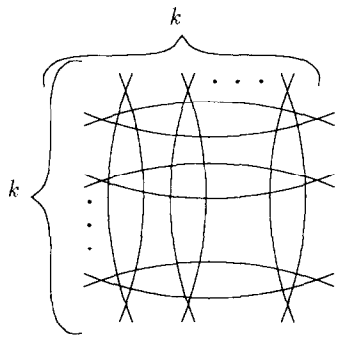

Fig. 4. The complexity of the region of all points at depth $2k + 2$ is $\Omega(k^2)$.

Of course, this argument does not really yield a lower bound of $\Omega(nk)$ for the kSC problem. We leave it as an open problem to find a model of computation in which such a lower bound can indeed be established. We note that there is some similarity between the construction just given and the recent construction of Erickson and Seidel [13] that proves a quadratic lower bound for the problem of testing whether a given set of lines in the plane contains three concurrent lines.

References

[1] P.K. Agarwal, J. Matoušek and D. Eppstein, Dynamic half-space reporting, geometric optimization, and minimum spanning trees, Proc. 33rd IEEE Sympos. Found. Comput. Science (1992) 80–89.

[2] P.K. Agarwal, M. Sharir and S. Toledo, New applications of parametric searching in computational geometry, J. Algorithms, to appear.

[3] A. Agarwal, H. Imai, N. Katoh and S. Suri, Finding k points with minimum diameter and related problems, J. Algorithms 12 (1991) 38–56.

[4] M. Ajtai, J. Komlós and E. Szemerédi, Sorting in $c \log n$ parallel steps, Combinatorica 3 (1983) 1–19.

[5] B. Chazelle, R. Cole, F.P. Preparata and C.K. Yap, New upper bounds for neighbor searching, Inform. and Control 68 (1986) 105–124.

[6] L.P. Chew and K. Kedem, Improvements on geometric pattern matching problems, in: Proc. 3rd Scand. Workshop Algorithm Theory, Lecture Notes in Computer Science 621 (Springer, Berlin, 1992) 318–325.

[7] R. Cole, Slowing down sorting networks to obtain faster sorting algorithms, J. ACM 34 (1987) 200–208.

[8] A. Datta, H.-P. Lenhof, C. Schwarz and M. Smid. Static and dynamic algorithms for k-point clustering problems, in: Proc. 3rd Workshop Algorithms Data Struct., Lecture Notes in Computer Science (Springer, Berlin, 1993) 265–276.

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

[10] A. Efrat and M. Sharir, A near-linear algorithm for the planar segment center problem, in: Proc. 5th ACM-SIAM Sympos, Discrete Algorithms (1994) 87–97.

[11] D. Eppstein, New algorithms for minimum area k-gons, in: Proc. 3rd ACM-SIAM Sympos, Discrete Algorithms (1992) 83–88.

[12] D. Eppstein and J. Erickson, New algorithms for minimum measure simplices and one-dimensional weighted Voronoi diagrams, Tech. Report 92-55, Dept. Inform. Comput. Sci., Univ. California, Irvine, CA, 1992.

[13] J. Erickson and R. Seidel, Better lower bounds on detecting affine and spherical degeneracies, in: Proc. 34rd IEEE Symp. on Foundation of Computer Science (1993) 528–536.
[14] K. Kedem, R. Livne, J. Pach and M. Sharir, On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles, Discrete Comput. Geom. 1 (1986) 59–71.
[15] H.P. Lenhof and M. Smid, Enumerating the k closest pairs optimally, Proc. 33rd IEEE Symp. on Foundation of Computer Science (1992) 380–386.
[16] D. Kirkpatrick and J. Snoeyink, Tentative prune-and-search for computing Voronoi vertices. in: Proc. 9th Annu. ACM Sympos. Comput. Geom. (1993) 133–142.
[17] T.C. Kao and D.M. Mount, An algorithm for computing compacted Voronoi diagrams defined by convex distance functions, in: Proc. 3rd Canad. Conf. Comput. Geom. (1991) 104–109.
[18] D. Leven and M. Sharir, Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams, Discrete Comput. Geom. 2 (1987) 9–31.
[19] J. Matoušek, On enclosing k points by a circle, Tech. Rept. KAM Series 93–245, Charles University, 1993.
[20] N. Megiddo, Linear-time algorithms for linear programming in R^3 and related problems, SIAM J. Comput. 12 (1983) 759–776.
[21] N. Megiddo, Applying parallel computation algorithms in the design of serial algorithms, J. ACM 30 (1983) 852–865.
[22] F.P. Preparata and M.I. Shamos, Computational Geometry, An Introduction (Springer, Berlin, 1985).
[23] R. Seidel, A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons, Comput. Geom. Theory Appl. 1 (1991) 51–64.
[24] M. Sharir, On k-sets in arrangements of curves and surfaces, Discrete Comput. Geom. 6 (1991) 593–613.
[25] C.K. Yap, An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments, Discrete Comput. Geom. 2 (1987) 365–393.