Enclosing k points in the smallest axis parallel rectangle

Article *in* Information Processing Letters · January 1996

DOI: 10.1016/S0020-0190(97)00212-3 · Source: CiteSeer

2 authors:

Michael Segal
Ben-Gurion University of the Negev

Klara Kedem
Ben-Gurion University of the Negev

Enclosing k points in the smallest axis parallel rectangle

Michael Segal and Klara Kedem*
Department of Mathematics and Computer Science
Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel

January 8, 1998

Abstract

We consider the following clustering problem. Given a set S of n points in the plane, and given an integer k, $\frac{n}{2} < k \leq n$, we want to find the smallest axis parallel rectangle (smallest perimeter or area) that encloses exactly k points of S. We present an algorithm which runs in time $O(n + k(n-k)^2)$ improving previous algorithms which run in time $O(k^2 n)$ and do not perform well for larger k values. We present an algorithm to enclose k of n given points in an axis parallel box in d-dimensional space which runs in time $O(dn + dk(n-k)^{2(d-1)})$ and occupies $O(dn)$ space. We slightly improve algorithms for other problems whose runtimes depend on k.

Keywords: Algorithms, computational geometry, axis-parallel, optimization.

1 Introduction

Given a set S of n points in the plane, and given an integer k, we want to find the smallest axis parallel rectangle (smallest perimeter or smallest area) that encloses exactly k points of S. This problem has been investigated by many researchers, some of whose results we cite below. They considered the problem for any $k \leq n$. Aggarwal et al. [2] present an algorithm which runs in time $O(k^2 n \log n)$ and uses $O(kn)$ space. Eppstein et al. [5] and Datta et al. [4] show that this problem can be solved in $O(n \log n + k^2 n)$ time; the algorithm in [5] uses $O(kn)$ space, while the algorithm in [4] uses $O(n)$ space. These algorithms are efficient for small k values, but become inefficient for large k's. Notice that for $k = n$ the smallest enclosing rectangle is trivially found in $O(n)$ time.

The algorithm which we present in Section 2 is more efficient than the ones cited above for k values in the range $\frac{n}{2} < k \leq n$. It is based on *posets* (partially ordered sets) [1]. It runs in time $O(n + k(n-k)^2)$ and $O(n)$ space. When $k = n$ our algorithm runs in $O(n)$ time. In Section 3 we extend our algorithm to higher dimensions and find the smallest axis parallel box that contains k out of given n points in d-space, $d \geq 3$. This algorithm runs in time $O(dn + dk(n-k)^{2(d-1)})$ and

*Work by Klara Kedem has been supported by a grant from the U.S.-Israeli Binational Science Foundation, and by a grant from the Israel Science Foundation founded by The Israel Academy of Sciences and Humanities.

occupies $O(dn)$ space. We assume that the rectangle (box) is closed, meaning that some of the k points can be on its boundary. We also assume that all the points of S are in general position, i.e., that no two points have the same coordinate in any axis. Finally, in Section 4 we shortly discuss slight improvements of other algorithms, when more efficiency is obtained by taking into account the size of k relative to n.

Remark. Another algorithm that runs efficiently for large k values was presented by Matoušek [7]. It finds the smallest circle enclosing all but few of the given n points in the plane. Given a large integer $k \leq n$ his algorithm runs in time $O(n \log n + (n-k)^3 n^{\varepsilon})$ for some $\varepsilon > 0$.

2 The Algorithm

In this section we present our algorithm for the planar problem. In Subsection 2.1 we describe an algorithm which finds the smallest enclosing rectangle that contains k x-consecutive points of S. The techniques used in this algorithm will be applied in our general algorithm, which is described in Subsection 2.2.

2.1 Enclosing k x-consecutive points

Given S as above, we restrict the problem to finding the smallest rectangle that covers k points of S whose x coordinates are consecutive. The x coordinate of an uncovered point of S is either among the $n-k$ smallest x coordinates or the $n-k$ largest ones. We cannot afford to spend $O(n \log n)$ time on sorting the points of S according to their x coordinates, therefore we apply a partial order selection method (see Aigner [1]). A *poset* is a partially ordered set of elements. Figure 1 below illustrates a poset S, where the largest $n-k+1$ points of S are sorted according to the order and the bottom $k-1$ points are known to be smaller but are not sorted.

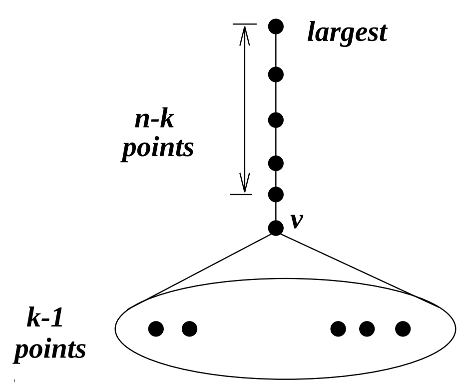

Figure 1: A poset

The construction of a poset where $R \subset S$ contains the $n-k$ elements of S with the largest x coordinates is easy. One way of doing this is to put n items into a binary heap and perform $n-k$ remove-max operations. In this way we collect the $n-k$ largest elements in S into an oredered set R in total time of $O(n+(n-k)\log n)$. We use this binary heap to find the point $v \in S$ with the $(n-k+1)^{th}$ largest x value in S.

Let $L = S - R$; clearly v is the point with the largest x coordinate in L (denoted by max_x^L). Denote by $x(v)$ ($y(v)$) the x- (y-) coordinate of v. We construct three binary heaps for L. They have k nodes each. We put the points of L into the heaps. The heap K_1 will be used to dynamically find the point with the smallest y coordinate in L (denoted by min_y^L). The heap K_2 will be used to dynamically find the point with the largest y coordinate in L (max_y^L), and D will help find the point with the smallest x coordinate in L (min_x^L). Finding the initial values above involves $3*(k-1)$ comparisons in the corresponding binary heaps.

Finding the rectangle

We slide a sweepline from left to right, starting at the leftmost point r of S. At this point we compute the perimeter (area) of the rectangle defined by min_x^L, max_x^L, min_y^L and max_y^L. The next event is to slide the sweepline to the next leftmost point of S: r is deleted from L, and v_1, the smallest point of R, is inserted into L, so that L always contains k points. The new max_x^L is $x(v_1)$. The next leftmost point in S is found using the binary heap D. This is the new min_x^L. We update the binary heaps K_1 and K_2, Thus we get the updated, possibly unchanged, min_y^L and max_y^L. Notice that we do not need to update D at all.

It is easily seen that each update takes $O(\log k)$ time, and the procedure is repeated $n-k$ times. Hence the total time involved in updates in $O((n-k)\log k)$. The initial construction of K_1, K_2 and D, is performed in total time of $O(n+(n-k)\log n)$.

Summing up the runtimes of constructing the heap and all the updates, we get

Theorem 2.1 *The smallest rectangle that contains a given number k, $\frac{n}{2} < k \leq n$, of x-consecutive points in a set of n points in the plane, can be found in time $O(n+(n-k)\log n)$.*

2.2 The smallest rectangle containing k arbitrary points

To avoid tedious notations we assume that the names of the points correspond to their x-ordering, though this does not mean that the points are sorted. In general the outline of our algorithm is as follows: initially we fix the leftmost point of the rectangle to be the leftmost point of S. At the next *stage* the leftmost point of the rectangle is fixed to be the second left point of S, etc. Within one *stage*, of a fixed leftmost rectangle point, r, we pick the rightmost point of the rectangle to be the q'th x-consecutive point of S, for $q = k + r - 1, \ldots, n$. For fixed r and q the x boundaries of the rectangle are fixed to be the x-coordinates of r and q respectively, and we go over a small number of possibilities to choose the upper and lower boundaries of the rectangle so that it will enclose k points.

In more detail, we initially produce the posets R, D, K_1 and K_2 as in the former algorithm. We use them as before but with a slight modification to the maintenance of K_1 and K_2 as we describe below. We also use two auxiliary *sorted* lists A_1 and A_2 that are initially set to be empty. They will collect the information found throughout the algorithm, of the lowest points (min_y^L) and highest points (max_y^L), respectively. The maximum size of A_1 and A_2 is $n - k$ each. Since the lists A_1 and A_2 are short we can afford $O(n - k)$ time update operation on them (search, insert, delete). As before, D and R are not updated throughout the algorithm.

For the initial rectangle (say $r = 1$ and $q = k$) we compute the perimeter (area) of the rectangle by the initial min_x^L, max_x^L, min_y^L and max_y^L. The point that attains min_y^L (max_y^L) is stored as the first element in A_1 (A_2).

For the next step, r remains fixed and $q = k + 1$, the vertical slab between r and q contains the first $k + 1$ x-consecutive points. Trivially there are two rectangles R_1 and R_2 containing k of these points within this slab that are defined by the x boundaries at r and at q. The y boundaries of R_1 are the second smallest y in K_1 and the first largest in K_2, and of R_2 the first smallest y in K_1 and the second largest in K_2. The second values found in K_1 and K_2 are stored in A_1 and A_2 respectively. We compute the area (perimeter) of these two rectangles and check for minimum.

Letting q vary from $k + 1$ to n, for each q we first update the data structures (see below) and then find the next smallest (largest) element in K_1 (K_2) and add it to the corresponding list A_1 (A_2). If $q = k + p$ then A_1 (A_2) has p entries, and we simply need to compute the areas of the rectangles bounded by r as min_x^L, q as max_x^L, and p min_y^L values from A_1 with their corresponding p max_y^L values from A_2.

Updating K_1 and A_2 upon varying q

1. If $y(q)$ is greater than the maximum y value in A_2, then no update of K_1 is required. This is because $y(q)$ will never get to act as $\min_y^L$ in the slab defined by r and q. We find the point with the maximum y value in A_2 by going over all its ($< n - k$) entries. We add q to A_2.

2. If $y(q) < \max(y)$ for the entries in A_2, then we can delete the point p which attains $\max(y)$ from K_1, and insert the point q into K_1. As in the former case p will not participate as a

lower y boundary of a rectangle in this slab. We remove the point p from A_2.

We update K_2 and A_1 symmetrically. Each heap update takes $O(\log k)$ time, and a list update takes $O(n-k)$ time. The heaps K_1 and K_2 remain of size k.

For each new *stage* ($r' = r+1$) we find the next smallest point (r') in S by removing the next minimal x point from the heap D. If r was in A_1 (A_2) we delete it. The heaps K_1 and K_2 undergo too many changes in stage r to be of any use at this stage. So we keep copies of the initial K_1 and K_2 from the previous stage r, and we only update them by deleting r and inserting $q = r+k-1$ instead of r in the heaps. (These will serve as initial K_1 and K_2 at the next stage.) We continue as in stage $r=1$, by incrementing q up to n and checking all the rectangles that contain k points between r and q. We finish when $r = n-k+1$ and $q = n$.

It is easy to see that we check all the rectangles that contain k points. Not all the rectangle possibilities in the above algorithm yield feasible rectangles. See, e.g., in Figure 2, where the rectangle whose x boundaries are determined by r and q, and the y boundaries are defined by the corresponding p^{th} points in A_1 and A_2. Checking whether a rectangle is feasible or not is immediate and does not change the complexity of the algorithm.

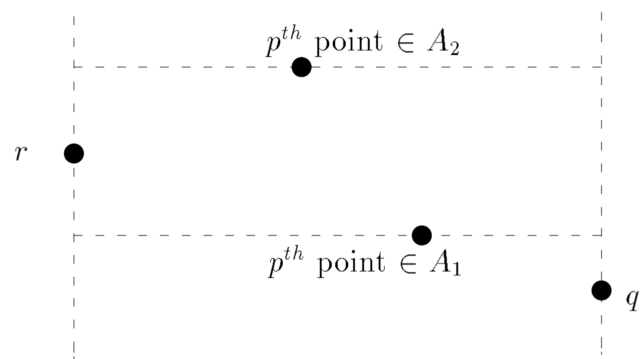

Figure 2: An infeasible rectangle

We sum up the runtimes of all the components of the algorithm:

- Computing R and initially constructing the heaps: $O(n + (n-k)\log n)$
- Copying the heaps K_1 and K_2 and initially updating them per each stage is: $O(k)$. For all stages $O(k(n-k))$.
- Total time for updating K_1, K_2, A_1 and A_2, for all the steps in one stage: $O((n-k)((n-k)+\log k))$. Summing up to $O((n-k)^2 \log k + (n-k)^3)$ for all the $n-k$ stages.

- The number of possible rectangles at each stage is bounded by the number of rectangles in the first stage: $\Sigma_{j=1}^{n-k} j = O((n-k)^2)$. Knowing A_1 and A_2 we invest $O(1)$ time in computing the area (perimeter) of each rectangle. The number of possible rectangles at all stages: $O((n-k)^3)$.

Since $k > n/2$ some of the above summands can be neglected and we yield

Theorem 2.2 *The smallest rectangle that contains a given number k, $\frac{n}{2} < k \leq n$, of points from a set of n points in the plane can be found in time $O(n + k(n-k)^2)$ and $O(n)$ space.*

3 The d-dimensional algorithm

We extend the planar algorithm to finding the smallest box containing k points in d-dimensional space. We first invest $O(dn)$ time in a preprocessing step that constructs the initial heaps described in Subsection 2.1, for each of the d axes. For $d \geq 2$ we denote by A_{d-1} the algorithm that finds the smallest box in $d-1$-space after the preprocess step. We denote its runtime by T_{d-1}. The d-dimensional algorithm is as follows. We project S on all the $d-1$-dimensional hyperplanes, call these sets $S_1, \ldots, S_d$. We describe an algorithm for one set, say S_i. (The process is repeated for all $S_j, 1 \leq j \leq d$.)

1. We use the algorithm A_{d-1} to find all the $d-1$-dimensional boxes that contain k to n points of S_i.

2. For each box found in the former step we use the i^{th} axis to bound exactly k points of S in the d-dimensional box which is the cross of the $(d-1)$-dimeansional box and a segment in the i axis (like we treated the y axis in the 2-dimensional problem).

It can be easily verified that the runtime of this algorithm is $(n-k)^2 T_{d-1}$. Adding the preprocessing time we get

Theorem 3.1 *The smallest box that contains a given number k, $\frac{n}{2} < k \leq n$, of points from a set of n points in d-space ($d \geq 3$) can be found in time $O(dn + dk(n-k)^{2(d-1)})$ and $O(dn)$ space.*

4 Slight improvements of other algorithms

We achieve improvements on runtimes of other problems that deal with some k-set problems under L_∞ metrics. For example, an algorithm for finding the minimum L_∞ diameter of a $k-$point subset of a set of n points in the plane is described in [5]. It runs in time $O(n \log^2 n)$. This algorithm can be improved to run in $O(n \log n \log (n-k))$ time for $k > \frac{n}{2}$. Eppstein and Erickson [5] use an $O(n \log n)$ time algorithm for placing a fixed-size axis-aligned square and then apply the technique of sorted matrices for the optimization step [6]. Applying our techniques we can solve the problem by dealing only with $(n-k)^2$ distances along each coordinate axis, instead of $O(n^2)$ distances as [5] do. Searching over this matrix adds a factor of $O(\log (n-k))$ instead of $O(\log n)$.

Recently, Glozman et al. [3] gave a simple algorithm for a problem posed (and solved) by Salowe [8]: Given a set S of n points in the plane, they [3, 8] determine, in time $O(n \log^2 n)$, which pair of points of S defines the k^{th} distance (smallest or largest) under the L_∞ metric. Both papers have the same decision algorithm, but for the optimization step [8] apply parametric search, while [3] apply sorted matrices. For $k \leq \frac{n}{2}$ it is enough to keep in the optimization matrix only $O(k^2)$ distances on each coordinate axis instead of all the $O(n^2)$. Thus the optimization will add only a factor of $O(\log k)$ instead of $O(\log n)$ as in [3].

References

[1] Martin Aigner, *Combinatorical search*, Wiley-Teubner Series in CS, John Wiley and Sons, 1988.

[2] A. Aggarwal, H. Imai, N. Katoh, S. Suri, "Finding k points with minimum diameter and related problems", *Journal of algorithms*, 12(1991),pp. 38–56.

[3] A. Glozman, K.Kedem, G. Shpitalnik, "On some geometric selection and optimization problems via sorted matrices", *Lecture Notes in Computer Science*, 955 (1995), pp. 26–35.

[4] A. Datta, H.-P. Lenhof, C. Schwarz, M. Smid, "Static and dynamic algorithms for k-point clustering problems", *Lecture Notes in Computer Science*, 709 (1993), pp. 265–276.

[5] D. Eppstein, J. Erickson, "Iterated nearest neighbors and finding minimal polytopes", *Disc. and Comput. Geom.*, 11(1994), pp. 321–350.

[6] G. Frederickson, D. Johnson, "Generalized selection and ranking: sorted matrices", *SIAM J. Comput.* 13(1984), pp. 14–30.

[7] J. Matoušek "On geometric optimization with few violated constraints", *Discrete Comp. Geom.* 14 (1995), pp. 365–384.

[8] J.S. Salowe "L-infinity interdistance selection by parametric search", *Inform. Proc. Let.*, 30(1989), pp. 9–14.