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

April 22, 1996

1 Introduction

Given a set S of n points in the plane, and given an integer k, $\frac{n}{2} \leq k \leq n$, we want to find the smallest axis parallel rectangle (smallest perimeter or smallest area) that encloses exactly k points of S. Aggarwal et al. [2] present an algorithm for this problem for any $k \leq n$, which runs in time $O(k^2 n \log n)$ and uses $O(kn)$ space. Eppstein et al. [4] and Datta et al. [3] show that this problem can be solved in $O(k^2 n)$ time; the algorithm in [4] uses $O(kn)$ space, while the algorithm in [3] uses linear space. These algorithms are quite efficient for small k values, but become inefficient for large k's. The naive algorithm, for $k = n$, will find the (only) rectangle in $O(n)$ time.

We present an algorithm which is more efficient than the algorithms cited above for large k values. It runs in time $O(n + k(n-k)^2)$ and uses linear space. Moreover, when $k = n$ our algorithm runs in $O(n)$ time.

We extend our planar algorithm to find a minimum volume axis parallel box in 3-space that contains k of n points from a point set S in the space. The $3d$ algorithm runs in time $O(n + k(n-k)^2 + (n-k)^5)$, and occupies $O(n)$ space.

2 The Algorithms

In this section we present our algorithms for the planar and $3d$ problems. In Subsection 2.1 we present an algorithm that finds the smallest enclosing rectangle that contains k x-consecutive points of S. This algorithm is used as a subroutine in our following algorithms. In Subsection 2.2 we present our algorithm to the planar problem stated in the Introduction, and in Subsection 2.3 the $3d$ algorithm. We assume that the rectangle (box) is closed, i.e., some of the k points can be on its boundary. We assume that all the points of S are in general position, meaning that no two points have the same x (or y, or z) coordinate.

2.1 Enclosing k x-consecutive points

We do not want to sort S according to x, since this will immediately spend $O(n \log n)$ time. Therefore we use a partial order selection method, described in Aigner [1], which creates posets for selecting the $n-k$ largest elements (x coordinate order) in S. A *poset* is a partially ordered set of

*Work by Klara Kedem has been supported by a grant from the U.S.-Israeli Binational Science Foundation.

elements. Figure 2 below illustrates a poset S, where the largest $n-k+1$ points are sorted and the bottom $k-1$ points are known to be smaller but are not sorted.

Let v denote the smallest point in the subset of the larger x coordinates. Denote by $x(v)$ the x coordinate of v. We are interested in the set of points $R \subset S$ whose x coordinates are larger than $x(v)$.

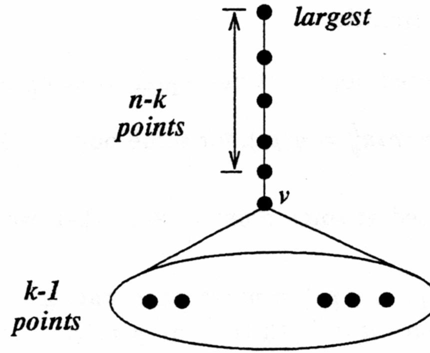

Figure 1: A poset

We describe how to find the poset R. We set a knock-out tournament as described in [1]. Assuming n is a power of 2 we put all the points of S in the leaves of a tree and compare them by pairs. As in a knock-out tennis tournament we determine the winner of each pair, put it in the parent node of the pair and repeat the process for this level: pair winners again and copy the winner of this stage to the parent level, and so on. The best player in the tournament, w, is determined by $n-1$ games and is at the root — zero level of the tree. We report w as the largest found and take it out of the tournament (put it in R). Take w's opponent from the first level in the tree and compare it to each of w's opponents along the path that w climbed up to the top (going from the root down to the leaf). After at most $\lceil \log n \rceil - 1$ games the second best player s is determined and reported (in R). Note that the third best player must have lost to the second best by this set-up. Hence the highest opponent of s now plays against all of s's opponents on the path that moved s to the top. It is immediately seen that in the worst case we need no more than $\lceil \log(n-1) \rceil - 1$ games to move the 3rd best player up to the top. Continuing in this way, we have collected the $n-k$ largest elements in S into an oredered set R, in time $O(k + \Sigma_{i=k+2}^{n} \lceil \log i \rceil) \leq O(k + (n-k) \log n)$.

We have R, the x-ordered subset of $n-k$ points of S with the largest x coordinates. Let $L = S - R$. We know v - the point with the largest x coordinate in L.

We perform three similar tournaments on L, but to a smaller extent. We find the point with the smallest y coordinate among the points in L, using the knockout strategy and resulting with a full binary tree K_1 (with k leaves) which stores the y coordinates of the points of L in the leaves, and the results of the intermediate tournaments in the inner nodes. Denote by min_y^L the winner in K_1. Similarly, we construct a tournament tree K_2 for finding the largest y coordinate, max_y^L. We repeat this for finding the smallest x coordinate, min_x^L, building a tournament tree D. K_2 and D are also trees of k leaves each. Notice that $max_x^L = x(v)$. We create an array U with n entries, an entry for each point $p_i \in S$. Each entry contains two pointers, one to the location of p_i in K_1 (assigned to be *nil* if p_i is not in K_1), and the other pointer for the location of p_i in K_2 (or *nil*).

Finding a 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 equals $x(v_1)$ and is found in time $O(1)$. The next leftmost point in S is found using the tournament step of finding the second winner in D. This is the new min_x^L. Finding the next leftmost point in D takes $\log k - 1$ comparisons.

We update the tournament trees K_1 and K_2 according to the deletion of the first point and the insertion of v_1. There are few possibilities:

1. The y coordinate of the deleted point r is not equal to min_y^L or max_y^L.

   - If $y(v_1) < min_y^L$, where $min_y^L = y(p_i)$, for some point $p_i \in L$, then we replace in K_1 the value $y(p_i)$ by $y(v_1)$.
   - If $y(v_1) > max_y^L$ attained at some point $p_j \in L$, then we replace in K_2 the value $y(p_j)$ by $y(v_1)$.
   - If $min_y^L = y(p_i) < y(v_1) < y(p_j) = max_y^L$ then find the location of the deleted point r in K_1 and K_2 and replace it in both trees by $y(v_1)$).

   We update the tree K_1 and the array U as follows: Using U we find the leaf containing $y(r)$ in K_1, replace it by $y(v_1)$, and moving from this leaf to the root of K_1, we compare $y(v_1)$ with the values in the adjacent nodes along the path and update K_1 accordingly. This step takes $O(\log k)$ time. We update U in time $O(1)$. We put the pointer to K_1 that was in entry r of U as the pointer to K_1 for entry v_1 in U. We put *nil* in the entry of r in U. Symmetrically we update K_2 and the pointers to K_2 in U.

2. The y coordinate of the deleted point r is equal to min_y^L or max_y^L. Say, wlog, to min_y^L.

   We find the second smallest y value in K_1 as we did in the previous step; we put $y(v_1)$ into the leaf of K_1 just vacated by min_y^L; and update the tree and U accordingly (as in step 1).

Notice that we do not need to update D at all. This procedure is repeated $n - k$ times. Hence the total time involved in updates in $O((n-k)\log k)$.

The construction of U takes $O(n)$ time. The initial construction of K_1, K_2 and D, is performed in total time of $O((k-1) + (\log k - 1) + (\log(k-1) - 1) + \ldots (\log(k - n + k - 2) - 1))$, which (substituting the first k in each log term by n) is $\leq O(\Sigma_{i=k+2}^{n} \lceil \log i \rceil) \leq O(k + (n-k)\log n)$.

Summing up the runtimes of the updates and constructing the trees, we get

Theorem 2.1 *The smallest perimeter (area) rectangle that contains a given number k of x-consecutive points in a set of n points in the plane, can be found in time $O(n + (n-k)\log(kn))$.*

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, and we go over a small number of possibilities to choose the upper and lower boundaries of the rectangle so that it encloses k points.

In more detail, we initially produce the posets R, D, K_1 and K_2 and the array U, 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 will 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, plus pointers to the leaves in K_1 and K_2 containing these points (*nil* as a pointer if the point is not the corresponding tree). The maximum size of A_1 and A_2 is $2(n-k)$ as will be seen below. Since the lists 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, and U's updates are done in $O(1)$ time.

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, $q = k+1$, the vertical slab between r and q contains the first $k+1$ x-consecutive points. Trivially there are two rectangles containing k of these points within this slab: (1) The second smallest (y) in K_1 (store it in A_1) and the first largest in K_2, and (2) The first smallest (y) in K_1 and the second largest in K_2 (store it in A_2).

For each *stage* (r) we use the tree D to find the next smallest point r in S. We go over all the steps within this stage, varying q (found in R) as above and finding the smallest rectangle that contains k points in the vertical slab determined by r and q.

As we go over the rectangles containing k points in the fixed vertical slab we add the newly found points with smallest (largest) y coordinates to A_1 (A_2) sorted by their y-value.

Not all these possibilities yield feasible rectangles. See, e.g., in Figure 2, where the rectangle determined by r and q contains k points that do not include q. But the infeasible rectangles are dealt with either in a previous step or in a subsequent step.

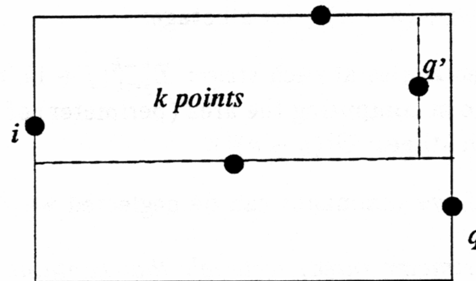

Figure 2: A rectangle with k points was dealt with in step q'

The main differences between the maintenance of K_1 and K_2 in this algorithm and the former are as follows. At the beginning of each stage r, the vertical slab determined by r and by $q = r+k-1$ contains k points in K_1 and K_2. As we take the next right point q, we add points to K_1. We will show below though, that the trees K_1 and K_2 remain of size k, but they undergo changes as q changes.

However, at the beginning of the next *stage* $r' = r+1$, $q' = q+1$, the new trees K_1 and K_2 will differ from the ones at the beginning of the former stage only by the deletion of r and insertion of q'. So we keep copies of these initial trees at stage r and update them by these two points to obtain the initial trees for stage $r+1$. Similarly, we have to keep a copy of the initial array U, and update it as we did above, for the deleted r and added q'. We initialize A_1 and A_2 to be empty at the beginning of the whole process. We add points to A_1 and A_2 throughout all the $(n-k)$ steps of stage $r = 1$. At the beginning of each stage $r' = r+1$ we delete from A_1 and A_2 the point r.

Assuming we have the initial trees, array and lists at the beginning of stage r and $q = r + k - 1$. For the next x-consecutive point q we do the following

Updating K_1 and A_2

- If $y(q) > \max(y)$ in A_2 for the entries in A_2 for which the pointer to K_1 is not *nil*, then we skip to the next q. This is because $y(q)$ will never be reached to act as $\min_y^L$ in the slab defined by r and q. We find the point with $\max(y)$ in A_2 by going over all its ($< n - k$) entries.

- If $y(q) < \max(y)$ in A_2 for the entries in A_2 for which the pointer to K_1 is not *nil*, then we can delete the point p_i which attains $\max(y)$ from K_1 and put the point q instead of it, updating the path to the root (in $O(\log(k)$ time). (as in the former case p_i will not participate as a lower y boundary of a rectangle in this slab.) We put *nil* in the pointer assigned to K_1 in A_2, for the entry p_i ($O(n - k)$ time).

Symmetrically we deal with K_2 and A_1.
Summarizing the runtime of our algorithm we get:

- Computing R: $O(k + \Sigma_{i=k+2}^{n} \lceil \log i \rceil) = O(k + (n - k) \log n)$

- Initially producing U: $O(n)$.

- Copying trees K_1 and K_2 and U and initially updating them per each stage is dominated by the latter: $O(n)$. For all stages $O(n(n - k))$.

- Total time for updating K_1, K_2, A_1 and A_2, for all the steps in one stage: $O((n - k)((n - k) + \log k)$. $O((n - k)^2 \log k + (n - k)^3)$ for all stages.

- The number of possible rectangles at each stage: $\Sigma_{j=1}^{n-k}(j + 1) = O((n - k)^2)$. 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 we yield

Theorem 2.2 *The smallest perimeter (area) rectangle that contains a given number k 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.*

2.3 The smallest box containing k arbitrary points in 3d-space

We extend the planar algorithm to the smallest box containing k points in the space. We project S on the x, y plane, call this planar set S_1.

1. We use the planar algorithm to find all the rectangles on the x, y plane that contain $k, k + 1, \ldots, n$ points of S_1. In essence we do the following:

2. For each rectangle found in the former step use the z axis to bound exactly k points of S in a $3d$ box defined by the x, y rectangle and a segment on the z axis.

(The whole process will be later repeated similarly for the x, z plane and a segment in the y axis and for the y, z plane and x axis.)

During the planar algorithm we encounter rectangles with k, $k+1, \dots, n$ points of S_1 in them. We construct additional tournament trees C_1 and C_2 for the z axis (similar to K_1 and K_2). Initially, for each vertical slab in the planar algorithm, the trees store the k minimal (maximal) values of the z-coordinates of the points projected in the rectangle. Similary to the arrays A_1 and A_2 in the planar algorithm, we construct and maintain two arrays T_1 and T_2 that save the points in increasing (decreasing) z order, that have been found during the algorithm. The vector U will now contain four fields per entry (for K_1, K_2, C_1, C_2). The new data structures are updated when the planar data structures are updated, and their update time is as for the corresponding planar data structures. For each rectangle generated in step 1 above, we go over the (length $n-k$) arrays, T_1 and T_2, and compute all the boxes determined by the z delimiter in $O(1)$ time per box.

Assume that we are at step m of stage r of the planar algorithm. There are $k' = k + m - r \geq k$ points of S_1 in the slab defined by r and m. We go over all the rectangles with $k, \dots, k'$ points in this slab. In the z dimension we perform an identical updating scheme as for the y direction in the planar algorithm, updating C_1, C_2, T_1, T_2 and U accordingly. The number of planar rectangles in one step is bounded by $(n-k)^2$, and in axis z by $(n-k)$, totalling in $(n-k)^3$ boxes per step.

As in the planar algorithm we check for consistency of the boundaries of the boxes This is done in constant time per box. We have $n-k$ stages with at most $n-k$ steps, so the running time of this algorithm is $O(k + k(n-k)^2 + (n-k)^5)$.

Remark

The same technique works when we have to deal with L_∞ metrics. For example, an algorithm for finding the minimum L_∞ diameter for a $k-$point subset of a set of n points in the plane is described in [4] and runs in time $O(n \log^2 n)$. Using the assumption that $k > \frac{n}{2}$ this algorithm can be improved to run $O(n \log n \log(n-k))$ time. They [4] used an $O(n \log n)$ time algorithm for placing a fixed-size axis-aligned hypercube and then applied the technique of sorted matrices described in [5]. Actually, we need not keep all the $O(n^2)$ distances along each coordinate axis in the matrix, but only $O((n-k)^2)$ (points that will be used as boundaries). Searching over this matrix adds a factor of $O(\log(n-k))$ and not $O(\log n)$ as in [4].

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, 38-56, (1991).

[3] A. Datta, H.-P. Lenhof, C. Schwarz, M. Smid, *Static and dynamic algorithms for k-point clustering problems* , In Proc. 3rd Workshop Algorithms Data Struct., pp. 265-276. Lecture Notes in Computer Science, vol.709. Springer-Verlag, New York, 1993.

[4] D. Eppstein, J. Erickson, *Iterated nearest neighbors and finding minimal polytopes*, Discrete Comput. Geom., 11, 321-350, (1994).

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