MAX-PLANCK-INSTITUT FÜR INFORMATIK

Finding k points with a smallest enclosing square

Michiel Smid

MPI–I–92–152 November 1992

Im Stadtwald
66123 Saarbrücken
Germany

Finding k points with a smallest enclosing square

Michiel Smid

MPI–I–92–152 November 1992

Finding k points with a smallest enclosing square

Michiel Smid*
Max-Planck-Institut für Informatik
W-6600 Saarbrücken, Germany
michiel@mpi-sb.mpg.de

November 27, 1992

Abstract

An algorithm is presented that, given a set of n points in the plane and an integer k, $2 \leq k \leq n$, finds k points with a smallest enclosing axes-parallel square. The algorithm has a running time of $O(n \log n + kn \log^2 k)$ and uses $O(n)$ space. The previously best known algorithm for this problem takes $O(k^2 n \log n)$ time and uses $O(kn)$ space.

1 Introduction

In statistical clustering and pattern recognition, the following problem often arises: Given a set of n points in the plane and an integer k, find k points that are similar, according to some similarity notion. (See Andrews [2] and Hartigan [4].) As an example, in [3], Dobkin et al. consider the problem of finding a convex polygon of smallest perimeter that contains k points. Aggarwal et al.[1] give algorithms for finding k points such that their diameter, or their enclosing square, or the perimeter of their enclosing rectangle is as small as possible.

In this paper, we consider the second problem studied in [1]. That is, we give an algorithm for finding a smallest axes-parallel square containing at least k points. Aggarwal et al. use k-th order Voronoi diagrams in the L_∞-metric to solve this problem. Their algorithm takes $O(k^2 n \log n)$ time and uses $O(kn)$ space. We solve the problem in $O(n \log n + kn \log^2 k)$ time using only $O(n)$ space, by means of a "double-sweep" algorithm. Hence, we improve the previously best known result by (roughly) a factor of k.

In Section 2, we consider a selection problem that we need later in the paper. Given two sets A and B of real numbers, we want to solve queries of the following form: Given a real number p and an integer k, find a k-th smallest element in the

*This work was supported by the ESPRIT Basic Research Actions Program, under contract No. 7141 (project ALCOM II).

set $A \cup (B + p)$, where $B + p := \{b + p : b \in B\}$. We also want to insert and delete elements in A and B. Section 2 gives a simple and efficient solution to this problem.

In Section 3, we solve a special case of our problem: We find a smallest square containing at least k points that has its left boundary on a fixed vertical line. This problem is solved by a plane sweep algorithm and it uses the data structure of Section 2. Then, in Section 4, we solve the complete problem. Again, we give a plane sweep algorithm. For each point encountered by the sweep-line, we use the plane sweep algorithm of Section 3 to check if the current optimal solution can be improved.

2 A selection problem

Let A and B be two sets of real numbers, and let $n := |A| + |B|$. Let $\preceq$ be an ordering on the real numbers, such that the relative ordering of any two reals can be computed in constant time. We want to store these sets such that for any real number p, and any integer k, $1 \leq k \leq n$, we can efficiently find a k-th smallest—w.r.t. $\preceq$—element in the set $A \cup (B + p)$, where $B + p := \{b + p : b \in B\}$. Moreover, it should be possible to insert and delete elements in A and B.

A k-th smallest element in A is said to be of *rank* k. An element of rank greater than $|A|$ is defined as ∞.

We maintain two balanced binary search trees T_A and T_B, storing the sets A and B, respectively, in sorted $\preceq$-order in their leaves. With each node in each of these trees, we store the number of leaves in its subtree. Hence, in A, we can find an element of any rank in $O(\log n)$ time. Also, given a real number p, we can find an element of any rank in the set $B + p$ in $O(\log n)$ time. Finally, we can insert and delete elements in A and B in $O(\log n)$ time.

In order to describe the query algorithm, we use the following notation: For $i \geq 0$, A_i denotes the set of the i smallest elements in A, i.e., the i leftmost elements in T_A. Note that $A_0 = \emptyset$ and $A_i = A$ for $i \geq |A|$. The set B_i is defined similarly.

Now consider a query: Let p be a real number and let k, $1 \leq k \leq n$, be an integer. Algorithm *select*(k, p) below finds an element of rank k in the set $A \cup (B + p)$. During the while-loop, it maintains the following

Invariant: i and j are integers, such that $0 \leq i, j \leq k$ and $i + j \geq k$. The set $A_i \cup (B_j + p)$ contains a k-th smallest element of $A \cup (B + p)$.

select(k, p):
begin
$i := k$; $j := k$;
while $i > 0$ and $j > 0$ and $i + j > k$
do $i' := \lfloor (i - j + k)/2 \rfloor$; $j' := k - i'$;
  let x be an element of rank i' in A;
  let y be an element of rank j' in B;
  if $x \preceq y + p$ **then** $j := j'$ **else** $i := i'$ **fi**
od;
if $i = 0$

then let z be an element of rank k in $B + p$
else if $j = 0$
    then let z be an element of rank k in A
    else (* comment: $i + j = k$ *)
        let x be an element of rank i in A;
        let y be an element of rank j in B;
        $z := \max(x, y + p)$
    fi
fi;
output the value of z
end

Lemma 1 *Algorithm select(k, p) finds an element of rank k in the set $A \cup (B + p)$. The running time is $O(\log n \log k)$.*

Proof: First we prove that the invariant is maintained during the while-loop. It is clear that the set $A_k \cup (B_k + p)$ contains a k-th smallest element of $A \cup (B + p)$. Therefore, the invariant holds after the initialization of the variables i and j. Consider one iteration and assume that the invariant holds at the start of it. Then $0 < i, j \leq k$, $i + j > k$ and the set $A_i \cup (B_j + p)$ contains a k-th smallest element of $A \cup (B + p)$.

It is straightforward to verify that $0 \leq i' \leq k$ and $i' + j \geq k$ for $i' = \lfloor (i - j + k)/2 \rfloor$. That is, if we set $i := i'$ in this iteration, then the first part of the invariant still holds. Similarly, the first part of the invariant is maintained if we set $j := j'$.

Let x and y be elements with ranks i' and j' in the sets A and B, respectively. Assume that $x \preceq y + p$. To prove that the second part of the invariant is maintained, we must show that the set $A_i \cup (B_{j'} + p)$ contains a k-th smallest element of $A \cup (B + p)$. Assume this is not the case. Let c be a k-th smallest element of $A \cup (B + p)$. Then, c must be contained in the set $(B_j \setminus B_{j'}) + p$. Note that all elements in $A_{i'}$ and $B_{j'} + p$ are at most equal to $y + p$, which itself is less than c. Therefore, there are at least $i' + j' = k$ elements in $A \cup (B + p)$ that are less than c. This is a contradiction. If $y + p \preceq x$, then it follows in the same way that the set $A_{i'} \cup (B_j + p)$ contains a k-th smallest element of $A \cup (B + p)$. This completes the proof that the invariant is correctly maintained.

Consider the quantity $i + j - k$. At the start of the algorithm, it has value k. It is easy to verify that $i' + j - k \leq (i + j - k)/2$ and $i + j' - k \leq (i + j - k + 1)/2$. That is, during each iteration, the quantity $i + j - k$ is reduced by a constant factor. By the invariant, it is always nonnegative. The while-loop terminates if $i + j - k$ is zero, or even earlier. Therefore, $O(\log k)$ iterations are made. In particular, the while-loop terminates.

After termination, there are three possible cases. If $i = 0$, then the invariant says that $B_j + p$ contains a k-th smallest element of $A \cup (B + p)$. Similarly, if $j = 0$, A_i contains such an element. Otherwise, $i + j = k$. Then, clearly, the maximal of the maxima of A_i and B_j is a k-th smallest element in $A \cup (B + p)$. Hence, the query algorithm is correct.

It remains to prove the running time. Each iteration takes $O(\log n)$ time. Hence, the while-loop takes $O(\log n \log k)$ time. After the while-loop is completed, it takes

another $O(\log n)$ time to compute the output. Hence, the entire query time is bounded by $O(\log n \log k)$. ■

The following theorem summarizes the result.

Theorem 1 *Let A and B be two sets of real numbers and let $\preceq$ be an ordering. There exists a data structure such that for any real number p and any integer k, $1 \leq k \leq n = |A| + |B|$, we can find a k-th smallest—w.r.t. $\preceq$—element in the set $A \cup (B + p)$ in $O(\log n \log k)$ time. Moreover, we can insert and delete elements in A and B in $O(\log n)$ time. The data structure has size $O(n)$ and can be built in $O(n \log n)$ time.*

3 A special case

Let S be a set of n planar points having non-negative x-coordinates and let k be an integer such that $2 \leq k \leq n$. Let p be a point of S. Consider the smallest square, if it exists,

1. whose left-side lies on the y-axis,
2. that has p on its bottom-side, and
3. that contains at least k points of S.

Let $l(p)$ denote the side-length of this smallest square. (If this square does not exist, then we define $l(p) = \infty$.) Our goal is an efficient algorithm that computes the minimal value of $l(p)$ over all points p in S. Notice that this is equivalent to computing a smallest square containing at least k points of S, whose left-side lies on the y-axis.

We can visualize the definition of $l(p)$ as follows. (See Figure 1.) Start with a square having its bottom-left and bottom-right corners at $(0, p_2)$ and p, respectively. Then, grow this square by moving the top-right corner under an angle of 45 degrees. Stop as soon as it contains at least k points of S. Then, $l(p)$ is the side-length of this final square.

Let S_p be the set of all points in S having a y-coordinate at least equal to p_2. We define an ordering $\leq_p$ on this set:

$$\text{For } a, b \in S_p\text{:} \quad a \leq_p b \quad \text{iff} \quad \max(a_1, a_2 - p_2) \leq \max(b_1, b_2 - p_2).$$

In geometric terms, a is at most equal to b in the ordering $\leq_p$, if in the process of growing the square, point b is not touched earlier than point a. Hence, if $a \in S_p$ has rank k w.r.t. this ordering, then the value of $l(p)$ is equal to the side-length of the corresponding square. Let

$$S_p^r := \{a \in S_p : a_1 \geq a_2 - p_2\},$$

and

$$S_p^t := S_p \setminus S_p^r = \{a \in S_p : a_2 - p_2 > a_1\}.$$

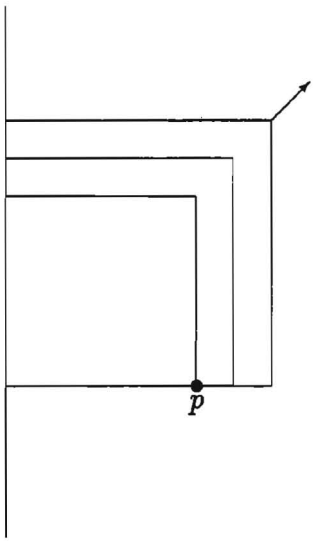

Figure 1: Determining $l(p)$ by growing a square.

In geometric terms, S_p^r contains all points of S_p that enter the growing square through its right side. Similarly, the points in S_p^t enter the growing square through its top side. Within the set S_p^r, the ordering by x-coordinates coincides with the ordering $\leq_p$. Similarly, within the set S_p^t, the ordering by y-coordinates coincides with the ordering $\leq_p$. An element of rank k, w.r.t. the ordering $\leq_p$, in the set S_p is the same as an element of rank k, again w.r.t. the ordering $\leq_p$, in the set $S_p^r \cup (S_p^t - p_2)$. Therefore, we can use the algorithm of the previous section, with $\preceq = \leq_p$, to compute $l(p)$.

We will compute the minimal value of $l(p)$ with a plane sweep algorithm. First, we discuss the relation of the orderings $\leq_p$ and $\leq_q$.

Lemma 2 *Let p and q be points of S, such that $q_2 \leq p_2$. Then, $S_p^t \subseteq S_q^t$.*

Proof: Let $a \in S_p^t$. Then $a \in S_p$ and $a_2 - p_2 > a_1$. Since $q_2 \leq p_2$, we also have $a_2 - q_2 > a_1$. Clearly, $a \in S_q$. Therefore, $a \in S_q^t$. ■

Lemma 3 *Let p and q be points of S, such that $q_2 \leq p_2$, and let $a \in S_p^r$. Then, $a \in S_q^t$ if and only if $a_2 - a_1 > q_2$.*

Proof: Assume that $a_2 - a_1 > q_2$. Since $a \in S_p^r \subseteq S_p \subseteq S_q$, it follows that $a \in S_q^t$. The converse follows from the definition of S_q^t. ■

We sweep over the points of S from top to bottom. During this sweep, we maintain the following information. Assume the (horizontal) sweep-line is at point p. Then the sets $A := S_p^r$ and $B := S_p^t$ are stored in a structure of Theorem 1. (The points in A and B are stored in increasing x- and y-order, respectively.) Moreover, we maintain a balanced binary search tree D, storing the elements of the set $\{a_2 - a_1 : a \in S_p^r\}$ in sorted order in its leaves. With each element $a_2 - a_1$, we store the name of the

corresponding point a. The root of D contains a pointer to the rightmost leaf, and the leaves are linked by pointers from right to left. Finally, a variable *best* contains the minimal l-value found so far.

At the start of the algorithm, p is the highest point in S. We initialize $A := \{p\}$, $B := \emptyset$ and $best := l(p) = \infty$.

Consider one iteration, in which the algorithm sweeps from point p to q. We do the following:

1. Insert q into A.
2. Insert $q_2 - q_1$ into D.
3. Walk along the leaves of D, from right to left. For each element $a_2 - a_1$ that is greater than q_2,
   (a) delete $a_2 - a_1$ from D;
   (b) delete a from A;
   (c) insert a into B.
4. Find an element of rank k, w.r.t. the ordering $\leq_q$, in $A \cup (B - q_2)$ and compute $l(q)$.
5. $best := \min(best, l(q))$.

If the algorithm has processed all points of S, the variable *best* contains the minimal value of $l(p)$ over all points p of S.

Note that the ordering $\preceq = \leq_q$ that is used in the algorithm depends on the point q. That is, for different q's, we get different orderings. It is easy to see, however, that this does not cause any problems.

Theorem 2 *The given algorithm finds the minimal element of* $\{l(p) : p \in S\}$, *in* $O(n \log n \log k)$ *time using* $O(n)$ *space.*

Proof: The correctness of the algorithm follows from the discussion above. Clearly, $O(n)$ space is used. Consider the running time. Each point of S is inserted once and deleted at most once in the set A. Moreover, each point is inserted at most once into B. Each difference $q_2 - q_1$ is inserted once and deleted at most once in D. Hence, the total time for updating A, B and D is bounded by $O(n \log n)$. For each point $q \in S$, we compute an element of rank k in the set $A \cup (B - q_2)$. By Theorem 1, this takes $O(\log n \log k)$ time per point. ∎

If $p \in S$ has the smallest l-value, then we can easily find k points that are contained in the corresponding square, within the given time and space bounds.

4 The final algorithm

We now give the algorithm that finds k points with a smallest axes-parallel enclosing square. This is equivalent to finding a smallest square that contains at least k points. Let S be a set of n points in the plane and let k, $2 \leq k \leq n$ be an integer. The following lemma is clear.

Lemma 4 *There is a smallest square containing at least k points, that contains points of S on its bottom- and left-side.*

For convenience, we add an $(n+1)$-st point with x-coordinate at $-\infty$. The points are stored in an array $X[0..n]$, in increasing x-order. The algorithm sweeps over the points from right to left. Assume the sweep-line is at point p. Let S_p be the set of points in $S \setminus \{p\}$ that have been visited already. A variable *smallest* is maintained, containing the best solution found so far, i.e., the side-length of a smallest square containing at least k points of S_p. All points of S_p that have a distance less than *smallest* to the sweep-line, are stored in a balanced binary search tree Y, in sorted y-order. Variables i and j are maintained, such that Y and $\{X[i], X[i+1], \ldots, X[j]\}$ contain the same points.

At the start of the sweep, we initialize $smallest := \infty$, Y as the empty tree and $i := j := n+1$.

Consider one iteration, in which the algorithm sweeps from point p to q. We do the following:

1. Insert p into Y.
2. Find the set V consisting of all points in Y that have a y-coordinate in the open interval $(p_2 - smallest : p_2 + smallest)$.
3. Use the algorithm of the previous section to find a smallest square whose left-side lies on the vertical line through p, that has a point of V on its bottom-side and that contains at least k points of V. Let *best* be the side-length of this smallest square.
4. $smallest := \min(smallest, best)$.
5. $i := i - 1$. Walk along X starting at $X[j]$. As long as the difference between the x-coordinate of $X[j]$ and q_1 is greater or equal to *smallest*, delete point $X[j]$ from Y and decrease j by one.

If the algorithm has visited all points of S, it outputs the value of the variable *smallest*.

To prove the correctness, consider one iteration of the algorithm. It is clear that only points in the set V can improve the current solution. Moreover, if the current solution is improved, then point p must be part of it. The algorithm of the previous section finds a best solution in the set V having its left-side on the vertical line through p. Clearly, if the addition of p leads to a better solution, the algorithm will

find it. Therefore, during each iteration, the information in *smallest*, Y, i and j are updated correctly.

Hence, the entire algorithm finds a smallest square that contains at least k points of S and that contains points of S on its bottom- and left-sides. By Lemma 4, this gives an optimal solution to our problem.

To analyze the running time, we need a bound on the size of the set V.

Lemma 5 $|V| \leq 2k - 1$.

Proof: The set V is contained in the rectangle

$$[p_1 : p_1 + \mathit{smallest}) \times (p_2 - \mathit{smallest} : p_2 + \mathit{smallest}).$$

This rectangle consists of two squares of side-length *smallest*. Each such square can contain at most $k - 1$ points of $V \setminus \{p\}$, because otherwise *smallest* was not the best solution in S_p. This proves that V has size at most $1 + 2(k - 1)$. ■

It takes $O(n \log n)$ time to build the array X. Each point of S is inserted once and deleted at most once in the tree Y. Hence, $O(n \log n)$ time is spent to update Y during the algorithm. Consider one iteration. The set V is found in time $O(\log n + |V|) = O(\log n + k)$. Then, the algorithm of the previous section is called for a set of size at most $2k - 1$. This takes $O(k \log^2 k)$ time. Therefore, the entire sweep algorithm takes time

$$O(n \log n + n(\log n + k) + nk \log^2 k) = O(n \log n + kn \log^2 k).$$

Clearly, the algorithm uses $O(n)$ space. The algorithm can easily be extended so that it also finds k points that are contained in the smallest square found, without increasing the complexity. This proves our final result:

Theorem 3 *Let S be a set of n points in the plane and let k, $2 \leq k \leq n$ be an integer. In $O(n \log n + kn \log^2 k)$ time, using $O(n)$ space, we can find k points of S with a smallest axes-parallel enclosing square.*

References

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

[2] H.C. Andrews. *Introduction to Mathematical Techniques in Pattern Recognition.* Wiley-Interscience, New York, 1972.

[3] D.P. Dobkin, R.L. Drysdale, III, and L.J. Guibas. *Finding smallest polygons.* In: Computational Geometry, Advances in Computing Research, Vol. 1, JAI Press, London, 1983, pp. 181-214.

[4] J.A. Hartigan. *Clustering Algorithms.* Wiley, New York, 1975.