ELSEVIER

Computational Geometry 8 (1997) 167–179

Computational Geometry
Theory and Applications

Translating a convex polygon to contain a maximum number of points ☆

Gill Barequet [a,*], Matthew Dickerson [b], Petru Pau [c]

[a] *School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv 69978, Israel*
[b] *Middlebury College, Middlebury, VT, USA*
[c] *University of Timişoara, Timişoara, România*

Communicated by J. Zaks; submitted 9 March 1995; accepted 1 May 1996

Abstract

Given a set S of n points in the plane and a convex polygon P with m vertices, we consider the problem of finding a translation of P that contains the maximum number of points in S. We present two different solutions. Our first algorithm uses standard line-sweep techniques and requires $O(nk \log(nm) + m)$ time where k is the maximum number of points contained. Our second algorithm requires $O(nk \log(mk) + m)$ time, which is the asymptotically fastest known algorithm for this problem. Both algorithms require optimal $O(m + n)$ space. The algorithms also solve in the same running time the bichromatic variant of the problem, where we are given two point sets A and B and the goal is to maximize the number of points covered from A while minimizing the number of points covered from B. © 1997 Elsevier Science B.V.

1. Introduction

A planar *translation* τ is an affine transformation of the plane that maps each point p in the plane to a point $\tau(p) = p + v$ for some fixed vector v. Given a translation τ and a polygon P, $\tau(P)$ is the polygon formed by the translation $\tau(p)$ of all points $p \in P$. We say that a polygon P *contains* a set S of points if every point in S lies on P or in the interior of P.

Problem 1. Given a convex polygon P and a planar point set S, find a translation τ that maximizes the number of points of S contained by $\tau(P)$.

☆ Work on this paper by the first author has been supported by the G.I.F., the German–Israeli Foundation for Scientific Research and Development, and by the Israeli Ministry of Science Eshkol grant 0562-1-94. Work by the second and third authors has been supported in part by NATO East Europe grant GER-93-55507. Work by the second author has been supported also by National Science Foundation grant CCR-93-01714.

* Corresponding author.

PII S0925-7721(96)00011-9

We present two algorithms to solve Problem 1. Our first algorithm improves on previous results of Efrat et al. [5] by an $O(\log m)$ factor, while the second more complicated approach improves the running time by an $O(\log n)$ factor.

1.1. Background

Problem 1 is one variation among many extensively researched clustering problems. For example, finding the smallest circle enclosing a given point set S is a famous problem in computational geometry (see [13, pp. 255–259]). This problem has been naturally extended to the smallest enclosing triangle [2,7,11], square, and rectangle [15], and the smallest enclosing convex polygon (which is the famous *convex hull* problem).

Another variant of this problem is: given a planar point set S and a fixed integer k, find a region that contains a *k-subset* of S and minimizes some measure such as area, radius, or circumference. Efrat et al. [5] give algorithms for computing the smallest k-enclosing circle and computing the smallest k-enclosing *homothetic* copy of a given convex polygon (of constant complexity). Eppstein and Erickson [4] also provide fast new solutions to a number of these problems including finding k-subsets of a given set S that minimize the following measures: area, perimeter, diameter, and circumradius.

A closely related problem is to find a placement of a region that maximizes the size k of the subset contained. That is, instead of fixing k and finding an optimal enclosing region, we fix the size and shape of the region and try to maximize k. This problem also has many of the same applications as the problems mentioned above and has been used as a substep in some of their solutions [4,5]. Problem 1 also has a strong similarity to various object recognition queries where we seek to match a pattern polygon to a query set of polygons, or a pattern point set to a subset of a query point set [6,10]. Problem 1 may be viewed as *matching* a pattern polygon to a query point set. Eppstein and Erickson [4], as a substep of their algorithm for finding the minimum L_∞ diameter k-subset of a given set S, note that an algorithm of Overmars and Yap [12] can be modified to find the maximum depth of an arrangement of axis-aligned rectangles. This approach solves in $O(n \log n)$ time the problem of finding an optimal translation of a rectangle to cover the maximum sized subset of S. That is, it solves Problem 1 in $O(n \log n)$ time in the special case when 'polygon' is a rectangle. Efrat et al. [5] as a substep in their algorithm for finding the smallest k-enclosing homothetic copy of an m-vertex polygon, claim an *oracle* solving Problem 1. They suggest a line-sweep technique for their oracle, but give no details. For the case when m is a constant, they claim the algorithm to run in $O(nk \log n)$ time, and for general m the complexity is worse by a factor of $O(\log m)$; that is, the algorithm requires $O(nk \log n \log m + m)$ time.

A variant of Problem 1 is the following problem.

Problem 2 (bichromatic coverage). Given a convex polygon P and two planar point sets A and B, find a translation τ such that the number of points in A contained by $\tau(P)$ minus the number of points in B contained by $\tau(P)$ is maximized.

1.2. Overview of new results and techniques

In this paper, we provide two general solutions to Problem 1 for arbitrary convex polygons. We first give details of a line-sweep algorithm, similar to that suggested in [5], and show that the algorithm

runs in $O(nk \log(nm) + m)$ time rather than $O(nk \log n \log m + m)$ time. We then present a second algorithm requiring $O(nk \log(mk) + m)$ time and $O(m + n)$ space, which is asymptotically faster than any previously known algorithms for this problem. We also show that the *bichromatic* version, Problem 2, can be solved in the same running time with only a slight modification of these algorithms. In fact, our algorithms solve a more general problem where all points have given weights, and the goal is to maximize the total weight of the contained points.

Both algorithms make use of a method for computing in $O(\log m)$ time the intersections between two translated copies of an m-vertex convex polygon. This method is based on *prune-and-search* techniques presented by Kirkpatrick and Snoeyink [8]. Our second algorithm is also based on a lemma that limits the number of possible translations to certain *translation stable* placements. A naive algorithm based solely on this lemma requires $O(n^2 m \log(mn))$ time. We show how to improve the complexity to output-sensitive $O(nk \log(mk) + m)$ time at no further cost in space. Our improvements are based on two techniques. The first technique relies on a property that relates translation stable placements to pairwise intersections of convex polygons, which may then be computed efficiently. The second technique is *bucketing*. Let A_P be the area of the smallest rectangle enclosing P, and let A_S be the area of the smallest rectangle enclosing S. In the case where the ratio A_S/A_P is $O(n)$, a bucketing approach using buckets of size A_P achieves the $O(nk \log(mk) + m)$ time bound at no further space cost. If A_S/A_P is $\omega(n)$, we use either a hash table to explicitly store only those buckets containing points from S, or a *degraded grid* approach as suggested in various papers by Lenhof and Smid [9]. The hash table method gives the same asymptotic running time in the expected case. The degraded grid approach uses a somewhat more complicated $O(n \log n)$ time preprocessing step, but gives an asymptotically equivalent worst case query time.

The outline of our paper is as follows. In Section 2 we present some definitions and lemmas upon which our algorithms are based. In Sections 3 and 4 we present our two main algorithms for Problem 1. In Section 5 we outline extensions of our algorithms to Problem 2. Section 6 provides our summary remarks and some open questions.

2. Geometric and algorithmic preliminaries

We now present some geometric results necessary for our algorithms. We begin with some definitions and notation used throughout the paper.

We use q_i to represent the ith point in our input set S. We assume that the polygon P is represented as a list of its vertices $p_1, \ldots, p_m$ given in clockwise order with p_1 located at the origin. Thus given a translation τ represented as a vector v, we can in constant time compute the position of the ith vertex of $\tau(P)$ as $p_i + v$ without explicitly computing the entire polygon.

We assume throughout the paper that the points in the set S are in *general position with respect to* P: no two points in S are on a line parallel to an edge of P.

We use the standard notation ∂P to represent the boundary of the polygon P. That is, ∂P is the union of the edges and vertices of P. Likewise, $\partial\tau(P)$ is the boundary of the translated polygon $\tau(P)$. We define a *translation stable placement* as follows. [1]

[1] This is similar to but different from the notion of a *stable placement* given by Chazelle [1]. Given two polygons P and Q such that P contains Q, Chazelle defines a *contact point* between P and Q as an intersection of a *vertex* of Q with an

Definition 1. Let $\tau(P)$ be a translation of polygon P containing a set of points S in general position with respect to P. We say that $\tau(P)$ is in *translation stable placement* if at least 2 points in S are on ∂P.

2.1. Limiting the search space

Chazelle [1] showed that if a polygon P contains a polygon Q, then there exists a rigid motion (translation and rotation) of P containing Q and in a stable placement. That is, at least one of the following conditions holds: (1) at least 3 points in S lie on ∂P; or (2) at least 2 points in S lie on ∂P and at least one point lies on a vertex of P. We may easily extend (or rather simplify) this result to show the following lemma.

Lemma 1. *Let S be a planar point set and let P be a convex polygon. If there is a translation τ such that $\tau(P)$ contains $k \geqslant 2$ points of S, then there exists a translation τ^* such that $\tau^*(P)$ also contains at least k points of S and is in translation stable placement.*

Actually, this result is stronger than we need. In our second algorithm, we limit our search space to translations τ with at least one point of S on $\partial\tau(P)$. However, we use the idea of stable placement to find these translations. The following lemmas show that given two points q_i and q_j and a polygon P, translation stable placements can be determined efficiently. (For Lemmas 2 and 3, see Fig. 1. For simplification and without loss of generality we let q_2 be on the origin $(0,0)$. That is, τ_2 is the null translation.)

Lemma 2. *Let P be a convex polygon, q_1, q_2 points, and τ_1 and τ_2 the translations mapping the origin to points q_1 and q_2, respectively. For any point x on ∂P, define $\tau_x = q_2 - x$ as the translation that maps x to q_2. Then $\tau_1(x)$ is a point of intersection between $\partial\tau_1(P)$ and $\partial\tau_2(P)$ if and only if q_1 is on $\partial\tau_x(P)$.*

Lemma 2 states that $\tau_x(P)$ is in translation stable placement with q_1 and q_2 on the boundary of $\partial\tau_x(P)$ if and only if $\tau_1(x)$ is a point of intersection between $\partial\tau_1(P)$ and $\partial\tau_2(P)$. The proof of this lemma follows from elementary geometry and vector arithmetic. In fact, the lemma easily generalizes to the following.

Lemma 3. *Let P be a convex polygon, q_1, q_2 points, and τ_1 and τ_2 the translations mapping the origin to points q_1 and q_2, respectively. For any point x, define $\tau_x = q_2 - x$ as the translation that maps x to q_2. Then $x \in (\tau_1(P) \cap \tau_2(P))$ if and only if $\tau_x(\tau_1(P))$ contains both q_1 and q_2.*

---

edge of P. When translations and rotations of P are allowed, a stable placement of polygons P and Q is one with three contact points. Thus in Chazelle's general definition, if a vertex of Q lies on a vertex of P, it intersects *two* edges of P and contributes two contact points. In contrast, we use a definition of stable placement that applies to translations only. Unlike Chazelle's original general definition, a translation stable placement is not defined by contact points but by two distinct points in S both of which are on $\partial\tau(P)$.

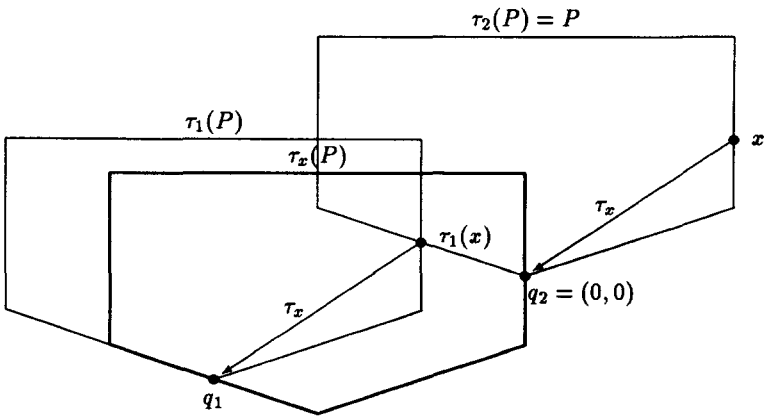

Fig. 1. Stable placements from intersecting polygons.

2.2. *Computing events quickly*

Both of our algorithms are based on event queues of some sort. In the first algorithm we use a standard line-sweep technique. In the second we use a different technique, that of an anchored sweep: sweeping the polygon around a point in S, keeping the polygon edges in contact with that point as we process events in clockwise order around the polygon. In both cases, as was shown in Lemmas 2 and 3, the events are determined by intersection points between polygons. Fortunately, we do not need to compute intersections between two arbitrary polygons but only between two translated copies of the same convex polygon. That is, we must solve the following problem.

Problem 3. Let P be a convex polygon with m vertices, and let τ_1 and τ_2 be translations. Compute the intersections (if any) between $\partial\tau_1(P)$ and $\partial\tau_2(P)$.

We show that this problem can be solved efficiently, as is stated in the following lemma.

Lemma 4. *Problem* 3 *can be solved in* $O(\log m)$ *time for translates of an* m*-vertex convex polygon.*

Proof. Consider the difference vector $\tau_\delta = \tau_2 - \tau_1$. At every point p on $\partial\tau_1(P)$ there is a corresponding point $p_\delta = \tau_\delta(p)$ on $\partial\tau_2(P)$. Thus if there is some point p on $\partial\tau_1(P)$ such that the point p_δ is also on $\partial\tau_1(P)$, then the point p_δ defines an intersection between $\partial\tau_1(P)$ and $\partial\tau_2(P)$. We thus reduce the problem of finding intersections between $\partial\tau_1(P)$ and $\partial\tau_2(P)$ to the problem of finding chords of $\partial\tau_1(P)$ of length and direction τ_δ.

This problem has been solved by Kirkpatrick and Snoeyink [8] using their "tentative prune-and-search" techniques for computing fixed-points. In [8], it is shown that the fixed-point of the composition of a monotone increasing piecewise-basic function and a monotone decreasing piecewise-basic function can be computed in $O(\log m)$ time. In [8, Section 3.1], they use this technique to solve the problems of computing the longest and specific chords parallel to a given direction α in $O(\log m)$ time. We actually require $O(m)$ time to preprocess the polygon into the necessary data structure, and $O(\log m)$ time per intersection query. □

2.3. Limiting more the search space

We have described how our search space can be limited to translation stable placements. We now show that it may be limited further to a number of events which is *output sensitive*. The method for doing this varies with the algorithm.

2.3.1. Bound on pairwise intersections

A more general version of the following result was proven by Sharir [14] and will be used in the analysis of our first algorithm.

Lemma 5 (Sharir). *Let $\mathcal{A}$ be an arrangement of n polygons in the plane having the property that the boundaries of each pair of polygons intersect at most twice. If the maximal depth of the arrangement is $\leqslant k$ then the number of intersections of pairs of the boundaries is* $O(nk)$.

2.3.2. The efficiency of bucketing

In our second algorithm we use bucketing to limit the number of pairs of points explicitly examined. To prove this method is efficient, we need two lemmas.

Lemma 6. *Let P be a convex polygon. There exist two rectangles R_P and R_I such that R_P encloses P and is no more than 2 times the area of P, and R_I is inscribed in P, is orthogonal to R_P, and is at least $1/2$ as long as R_P and at least $1/4$ as wide as R_P.*

Proof. Consider any diameter d_P of P. Let d be the length of d_P. Note that d_P divides P into two polygons (with one polygon possible empty, if d_P is actually an edge of P itself). In each of these polygons, we inscribe a triangle of maximum height with d_p as the base. Let a_1 and a_2 be the heights of these triangles. Their combined area is $\frac{1}{2}d(a_1 + a_2)$, and thus the area of polygon P is at least $\frac{1}{2}d(a_1 + a_2)$. To construct R_I, we let two vertices fall on d_p and use as the other two vertices the midpoints of the other two edges of the larger of these two triangles. Thus the length of R_I is exactly $\frac{1}{2}d$ and the height of R_I is at least $\frac{1}{4}(a_1 + a_2)$. Finally, we construct a rectangle R_P enclosing P with two sides parallel to d_P and of length d, and with two sides of length $a_1 + a_2$. The properties of R_P and R_I follow directly from our construction.[2] □

This result provides the necessary tool for the proof of the following lemma.

Lemma 7. *Let S be a point set and P a convex polygon. Then there exists a rectangle R_P, as in Lemma 6, enclosing P with the following property: if there exists a translation $\tau(R_P)$ of R_P containing k points of S, then there exists a translation $\tau^*(P)$ of P containing $\Omega(k)$ points of S.*

The proof of this lemma follows directly from Lemma 6.

[2] We note that Lemma 6 implies a maximum of 2 for the ratio of the area of the smallest enclosing rectangle to the area of the enclosed polygon. This is a tight bound, achieved if the polygon P is any triangle. We believe that the ratios for the R_I are not tight, but the proof is simple and is sufficient for our asymptotic results.

3. The first approach

We now provide our first algorithm for the solution to Problem 1. It is based on a technique now standard in computational geometry: the line sweep. We make use of the following observation: if a polygon P is reflected around a point p_i to form a new polygon P^{R}, then for any translation τ, $\tau(p_i)$ is contained by P^{R} if and only if p_i is contained by $\tau(P)$. Thus computing an optimal translation of P (the translation containing the maximum number of points) is equivalent to computing a location of maximum depth in the arrangement of polygons formed by translating a copy of the reflected P^{R} to every point $q_i \in S$. Efrat et al. [5] suggested this approach as an *oracle* in a parametric algorithm for computing the minimum area homothetic copy of a polygon containing k points for some fixed k. Without giving details, they claimed the algorithm has a running time of $\mathrm{O}(nk \log n)$ if m is constant, with an additional $\log m$ factor otherwise. (There is also an additive $\mathrm{O}(m)$ term for preprocessing of the polygon P; this is often omitted as m is assumed smaller than n.) We give details of this approach now, and prove a tighter bound of $\mathrm{O}(nk \log(nm) + m)$ rather than $\mathrm{O}(nk \log n \log m + m)$.

Our algorithm makes use of only two data structures, both of which are simple, standard, and easy to implement. The first structure is the event queue itself, used for the line sweep. It returns the next event ordered by x-coordinate. This is implemented as a standard priority queue. The **Add**(e, Q) operation adds an event e to the queue Q, and the **DeleteMin**(Q) operation returns the next event e from Q. Both operations require $\mathrm{O}(\log q)$ time, where q is the current size of the queue. The second structure keeps track of the current polygon chains. These are stored in a balanced binary tree, ordered by the y-coordinate at the current x position in the line sweep. We call this structure a *chain tree*. The **Add**(C, CT) operation adds a polygon chain C to a chain tree CT. The **NextEvent**(C, CT) operation returns the next intersection between a chain C and a neighboring chain in the tree CT. There is also a depth associated with the region between each pair of chains in the tree. The goal of the algorithm is to find the region of greatest depth.

3.1. Events

We describe the algorithm by first specifying the events in the queue. There are three main types of events.

Type 1. *First Vertex.* The first (leftmost) vertex in a polygon is the first type of event. There are n of these events, all computed and added to the queue in the initial stage. For every event of this type, we compute two chains C_l and C_u, the lower and upper chains leading from the leftmost to the rightmost vertex of the polygons. Both chains are added to the chain tree (where they are initially consecutive) and each chain's first intersection event is computed using **NextEvent** and added to the event queue. The depth of the new region between C_l and C_u is one greater than the depth of the region into which the initial point of the consecutive chains was inserted. The original region is split into two regions.

Type 2. *Last Vertex.* Likewise, the final (rightmost) vertex of a polygon also forms an event. This event ends two polygon chains C_l and C_u. Here a region ends, and the neighbor regions on both sides are merged to form a single new region. In doing so, a new pair of chains—call them C_{ll} and C_{uu}—becomes adjacent in the tree. We therefore must check **NextEvent**(C_{ll}, CT) and **NextEvent**(C_{uu}, CT) and add them to the event queue.

Type 3. *Chain Intersection.* A third and final type of event is the intersection of two polygon chains C_1 and C_2. In this case, the two chains swap order in the tree, and each must be checked using **NextEvent**. What happens to the depth of the regions may be divided into three subcases. If two upper chains intersect or two lower chains intersect, the depths of the regions remain the same. If an upper chain intersects with a lower chain, then the depth of the region increases or decreases by 2: in case the upper chain had lower y-value before the intersection and a higher y-value after the intersection, the depth of the region increases by 2; and in case the lower chain had lower y-value before the intersection and a higher y-value after the intersection, then the depth of the region decreases by 2.

We thus have the first algorithm.

ALGORITHM 1

I. Preprocessing: Let p_i be the rightmost vertex of P. Form P^{R} by reflecting P around p_i. Initialize a priority queue Q by adding every point $q_i \in S$ to the queue as an event of type 1.

II. Line Sweep:

1. **WHILE** not Empty(Q) **DO BEGIN**
2. Remove and process events as described in Section 3.1.
3. **END WHILE**

3.2. Analysis

There are n events of type 1 and n events of type 2. By Lemma 5, the number of events of type 3 is $O(nk)$. It follows that the event queue can be managed at a cost of $O(\log(nk))$ time per event. Since $k \leqslant n$, this is $O(\log n)$. Likewise, there are at most $2n$ active chains at any time, and so operations on the trees of chains also require $O(\log n)$ time. This leaves only the computation of intersection events. Lemma 4 showed that intersections of two translations of a convex polygon can be computed in $O(\log m)$ time. We can modify this algorithm so that it only reports intersections on the specified subchains of the polygons. We thus have $O(nk)$ events requiring $O(\log n + \log m)$ time per event for a total running time of $O(nk \log(nm))$ plus $O(m)$ time to preprocess the polygon P.

4. The second approach

We now present a second alternative and conceptually somewhat different algorithm for the solution to Problem 1. From Lemma 1, we see that we may limit our search to translations of the polygon which are in translation stable positions. A naive approach based on this Lemma is, for every edge $e_i \in P$ and for every point $q_j \in S$, translate e_i onto q_j, and then slide e_i along q_j in discrete intervals determined by the translation stable placements. For each of the $\Theta(nm)$ edge-point pairs, we need to compute the distance of every other point in S to the boundaries of the current translated polygon in the direction determined by e_i, and keep an updated event queue. Based on the results presented in Section 2, this approach can be made to run in time which is asymptotically faster than Algorithm 1 when k is "small".

I. Preprocessing: Preprocess points into buckets. Initialize Q.

II. Iteration

1. Set max := 0. {Maximum # of points contained so far}
2. **FOR** each point $q_i \in S$ **DO BEGIN** {Anchored sweep from every point}
3. Set $c := 1$. {Points contained by current translation}
4. **FOR** each $j \neq i$ and $q_j \in B_i$ **DO BEGIN** {Examine nearby points for containment}
5. Compute intersections of $\partial\tau_i(P)$ and $\partial\tau_j(P)$. {Compute stable placements with q_i, q_j}
6. Let $\tau_j(x)$ be a discrete intersection point; ADD (x, j) to Q.
7. **IF** q_j is contained by $\tau_i(P)$
   THEN Mark q_j "IN"; Set $c := c + 1$;
   ELSE Mark q_j "NOT IN".
   END IF
8. **END FOR**
9. **WHILE** $Q \neq \emptyset$ **DO BEGIN** {Sweep with stable placements as events}
10. Delete (x, j) from front of Q. {Update structures and counters}
11. **IF** q_j is "NOT IN"
    THEN Set $c := c + 1$; Mark q_j "IN".
    ELSE Set $c := c - 1$; Mark q_j "NOT IN".
    END IF
12. **IF** $c >$ max, **THEN** Set max := c; Store translation. **END IF**
13. **END WHILE**
14. **END FOR**

Fig. 2. Algorithm 2.

First, we use bucketing as follows to limit the number of pairs of points examined. Let R_P be a rectangle enclosing P and of area proportional to P (as described in Lemma 6). Let R_S be the smallest-area rectangle orthogonal to R_P and enclosing S. We use R_P to partition R_S into a grid of "buckets", and then place each point of S into its appropriate bucket. Note that for a given point $q_i \in S$, there are at most 9 buckets intersected by all polygons $\tau(P)$ with q_i on its boundary. We define the *neighborhood* B_i of point q_i as the bucket that q_i is in plus its 8 adjacent buckets, including those diagonally adjacent. Our search from point q_i will be limited to points in B_i.

Second, we avoid recomputing distance information for all points for every edge e_i of P. Instead, we compute for all $j \neq i$ and $q_j \in B_i$ at once all stable placements between q_i and q_j using Lemmas 4 and 7. The resulting algorithm is given in Fig. 2. We use τ_j for the translation mapping the origin to the point $q_j \in S$. We let Q be a priority queue of pairs (x, j), where x is a point on ∂P and the points are ordered in clockwise order around P. We can represent x by the edge number and the distance along the edge. The proof of correctness follows from the results of Section 2.

4.1. Analysis

We now present an asymptotic analysis of the time required by Algorithm 2, given in Fig. 2.

The outer loop beginning at Step 2 is iterated n times. It follows from Lemma 7 that for each iteration of the outer loop, the inner loop beginning at Step 4 is iterated $O(k)$ times. So Steps 5 through 7 in the inner loop are iterated $O(nk)$ times. From Lemma 4, we see that the intersections at

Step 5 can be computed in $O(\log m)$ time. Likewise, the polygon inclusion queries of Step 7 may be answered in $O(\log m)$ time.

Assume general position, such that we have at most two discrete intersections per polygon pair. The priority queue for each point thus has at most $2k$ events. It follows that the number of queue events in the loop beginning at Step 9 is also $O(k)$ and that each queue operation requires $O(\log k)$ time. The algorithm therefore requires a total of $O(nk(\log m + \log k) + m) = O(nk \log(mk) + m)$ time if an appropriate bucketing strategy is used.

Though bucketing is often used in expected case analyses, it is important to note that we are analyzing our use of bucketing for a *worst case* time bound. Though the number of points in a particular bucket may grow as large as $\Theta(n)$, by Lemma 7, k is asymptotically as large as the number of points in the densest bucket. That is, the running time is sensitive to k. We also note that only minor modifications are required for relaxing the general-position assumption. When two polygons intersect along an edge, only the initial point on the first edge and the final point on the second edge need be added to the queue. Both of these are polygon vertices.

4.1.1. A note on bucketing

The drawback to this approach is that the number of buckets does not depend on n, m or k but on A_S/A_P, the ratio of the area of the smallest rectangle enclosing S to the area of P. The initialization step requires $O(n + m + A_S/A_P)$ time and space. In the case where the polygon is very small relative to the space occupied by our point set S, our bucketing algorithm is no longer efficient. We would like an algorithm that is fast in this case as well. There are two different approaches that will resolve the problem. The first solution is a common one in computational geometry: instead of explicitly computing, initializing, and storing all the $\Theta(A_S/A_P)$ buckets, we only store those buckets that actually contain points. This may be accomplished using a hash table of $O(n)$ buckets. Every bucket reference is resolved with a lookup to this $O(n)$ sized hash table, which can be done in $O(1)$ expected case time. Thus the running time is still $O(nk \log(mk) + m)$ but in the expected case.

A second solution is to use the so-called *degraded grids* introduced by Lenhof and Smid [9]. This approach achieves the same asymptotic running time as hash tables but in the worst case rather than expected case. The idea is to use varying sized buckets whose size is the same as R_P if they contain a point but which grow maximally long and maximally wide if empty. This ensures that the total number of buckets is still $O(n)$. The cost of this approach is a more complicated $O(n \log n)$ time preprocessing step requiring that the input set S be sorted. We note however that this sorting step needs to be done only once for S. Repeated queries on S can be made with different polygons without repeating the $O(n \log n)$ time preprocessing. The reader is referred to the paper cited above for details on this elegant approach.

We note that for practical purposes the buckets may often be implemented using rectangles orthogonal to the coordinate axes. If the smallest orthogonally oriented rectangles have areas of the same magnitude as the R_P described in the algorithm, then this will not affect the asymptotic running time of the algorithm and could simplify implementation. Similarly, the rectangles used for bucketing may be enlarged or shrunk to lie on more "natural" boundaries for bucketing. Increasing the size of the bucket will decrease the number of buckets that need to be examined per point, but will increase the number of points per bucket. The algorithm may be fine-tuned by adjusting the actual bucket size.

5. Weighted and bichromatic sets

The algorithms presented here can easily be extended to a more general version of the problem. Consider a set S where each point q_i is given a weight $W(q_i)$. Instead of maximizing the number of points in S covered by $\tau(P)$, we want to maximize the total weights of all points covered. For example this could be used to solve the so-called *bichromatic* version of the problem where the objective is to maximize the number of points contained from one set, say the *red* set, while minimizing points contained from another set, say the *blue* set. One set of points is given positive weights while the other is given negative weights.

If $\forall i$: $1 \leqslant i \leqslant n$ $W(q_i) \geqslant 0$, then Lemma 1 still applies and the algorithms run with no modifications. However, if $W(q_i) < 0$ for some points in the set, as in the bichromatic problem, then it is possible that there is *no* translation that maximizes the total weight of the points covered and is still in translation stable placement. (It is not difficult to give an example where the only stable placement covering the same points as covered by the maximal translation also covers an additional negatively weighted point.) The solution to this problem is not difficult and applies to both algorithms. For each translation τ with a negatively weighted point q_i on the boundary, we look for a nearby translation τ_ε that contains the same points but does not contain q_i. The same idea can be applied in the degenerate case where multiple points lie on the boundary, though if there are more than one negatively weighted points on the boundary then the ε translation will not necessarily exist. Thus with minor modifications, both Algorithms 1 and 2 can be used to solve Problem 2 in the same running times.

6. Summary

We have provided two asymptotically fast solutions to Problem 1: computing a translation of a given polygon P that contains the maximum number of points of a given point set S. The faster of the two algorithms requires $O(nk \log(km) + m)$ time which is asymptotically faster than all previously known solutions. The algorithms are conceptually simple, using either a line sweep or an anchored sweep. They are also self-contained except for the use of the tentative prune-and-search technique of Kirkpatrick and Snoeyink [8] for computing intersections of the polygons in $O(\log m)$ time, beating the more straightforward nested binary search which would require $O(\log^2 m)$ time. The algorithms also generalize at no cost in running time to the bichromatic variant of the problem, and also to the more general weighted point set problem.

6.1. Extensions and open problems

There are three obvious generalizations of Problem 1. We may consider containment by translation of arbitrary simple polygons, containment by rigid motion (translation and rotation) of either simple or convex polygons, or containment by polyhedra in higher dimensions.

Problem 4. Given a simple polygon P and a planar point set S, find a translation τ that maximizes over all possible planar translations τ the number of points of S contained by $\tau(P)$.

Problem 5. Given a convex polygon P and a planar point set S, find a rigid motion τ that maximizes over all possible planar rigid motions τ the number of points of S contained by $\tau(P)$.

Problem 6. Given a convex polyhedron P a point set S in $\mathbb{R}^3$, find a rigid motion τ that maximizes over all possible rigid motions τ the number of points of S contained by $\tau(P)$.

Both algorithms presented here may be used with some modifications to solve Problem 4. However, the algorithms are significantly less efficient. First, the number of pairwise intersections between a single pair of m-vertex simple polygons may be $\Theta(m)$ instead of 2. Also, the $O(nk)$ bound on pairwise intersections no longer holds. So the number of intersection events in the event queue in Algorithm 2, for example, can grow to as much as $\Theta(n^2m)$. Second, the intersections can no longer be computed efficiently using the method discussed in Section 2. Finally, bucketing is no longer guaranteed to save time, as Lemma 7 does not hold for simple polygons. This suggests a much less efficient algorithm requiring $O(n^2m\log(mn))$ time to solve Problem 4.

When we allow rotation as well as translation, then even for convex polygons the problem becomes yet more complicated. Chazelle [1] has shown that all general stable placements of an m-vertex polygon P on three points can be computed in $O(m^2)$ time. This leads to an algorithm to solve this problem for general rigid motion. For each triple of points in S, compute all stable placements of P, and for each placement count the number of points contained. This requires $O(n^4m^2\log m)$ time. This is impractical for even moderately large point sets.[3]

Problem 7 (open). Determine the lower bound for Problem 1.

Problem 8 (open). Give an efficient algorithm for Problem 4. (A brute force line-sweep approach in which we explicitly compute all n rotated polygons, and all pairwise intersections, yields an algorithm requiring $O(n^2m\log(mn))$ time.)

Acknowledgements

This paper benefited from discussions with Scot Drysdale, Dan Halperin, Matthew Katz, and others at the 10th ACM Symposium on Computational Geometry, Stony Brook NY, June 1994, and also from a discussion with Jack Snoeyink.

References

[1] B. Chazelle, The polygon placement problem, in: F. Preparata, ed., Advances in Computing Research 1 (JAI Press, 1983) 1–34.

[2] S. Chandran and D. Mount, A parallel algorithm for enclosed and enclosing triangles, Internat. J. Comput. Geom. Appl. 2 (2) (1992) 191–214.

[3] M. Dickerson and D. Scharstein, Optimal placement of convex polygons to maximize point containment, in: Proc. 7th ACM–SIAM Symp. Discrete Algorithms 1996, to appear.

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

[5] A. Efrat, M. Sharir and A. Ziv, Computing the smallest k-enclosing circle and related problems, Computational Geometry 4 (1994) 119–136.

[3] Dickerson and Scharstein [3] have recently claimed a solution to this problem requiring $O(nk^2\log(mn))$ time.

[6] D.P. Huttenlocher and S. Ullman, Object recognition using alignment, in: Proc. 1st Internat. Conf. Computer Vision (IEEE Computer Science Press, 1987) 102–111.
[7] V. Klee and M.L. Laskowski, Finding the smallest triangles containing a given convex polygon, J. Algorithms 6 (1985) 359–375.
[8] D. Kirkpatrick and J. Snoeyink, Tentative prune-and-search for computing fixed-points with applications to geometric computing, in: Proc. 9th ACM Symp. Comput. Geom. (1993) 133–142.
[9] H.P. Lenhof and M. Smid, Sequential and parallel algorithms for the k closest pairs problem, Internat. J. Comput. Geom. Appl. 5 (1995) 273–288.
[10] Y. Lamdan, J.T. Schwartz and H.J. Wolfson, Object recognition by affine invariant matching, in: Proc. IEEE Conf. on Computer Vision and Pattern Recognition (IEEE Computer Science Press, 1988).
[11] J. O'Rourke, A. Aggarwal, S. Maddila and M. Baldwin, An optimal algorithm for finding minimal enclosing triangles, J. Algorithms 7 (1986) 258–269.
[12] M.H. Overmars and C.K. Yap, New upper bounds in Klee's measure problem, SIAM J. Comput. 20 (1991) 1034–1045.
[13] F.P. Preparata and M.I. Shamos, Computational Geometry: An Introduction (Springer, Berlin, 1985).
[14] M. Sharir, On k-sets in arrangements of curves and surfaces, Discrete Comput. Geom. 6 (1991) 593–613.
[15] G.T. Toussaint, Solving geometric problems with the rotating calipers, in: Proc. IEEE MELECON (1983).