1990

Computing Some Distance Functions Between Polygons

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Celso C. Ribeiro

Sergio Lifschitz

Report Number:
90-1006

COMPUTING SOME DISTANCE FUNCTIONS BETWEEN POLYGONS

Mikhail J. Atallah
Celso C. Ribeiro
Sergio Lifschitz

CSD-TR-1006
August 1990
(Revised September 1990)

Computing Some Distance Functions Between Polygons

Mikhail J. Atallah*
Purdue University
Department of Computer Science
West Lafayette, IN 47907
USA

Celso C. Ribeiro†
GERAD and École Polytechnique de Montréal
5255, avenue Decelles
Montréal (Québec)
Canada H3T1V6

Sergio Lifschitz
Catholic University of Rio de Janeiro
Department of Electrical Engineering
Caixa Postal 38063 - Gávea
Rio de Janeiro 22452
Brazil

May 1990

Abstract

We present algorithms for computing some distance functions between two (possibly intersecting) polygons, both in the convex and nonconvex cases. The interest for such distance functions comes from applications in robot vision, pattern recognition and contour fitting. We present a linear sequential algorithm and an optimal EREW-PRAM parallel algorithm for the case when the input polygons are convex, and an essentially quadratic sequential algorithm for the case of arbitrary polygons (possibly with holes).

Keywords: Computational geometry, polygons, distance computation, Hausdorff distance, pattern recognition, contour fitting, robot vision, algorithms.

*This author was supported by the Office of Naval Research under Contracts N00014-84-K-0502 and N00014-86-K-0689, the Air Force Office of Scientific Research under Grant AFOSR-90-0107, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118.

†This author is on leave from the Catholic University of Rio de Janeiro, Department of Electrical Engineering, Caixa Postal 38063, Rio de Janeiro 22452, Brazil. His work was partially supported by *Conselho Nacional de Desenvolvimento Científico e Tecnológico* under Grant 202005/89.5

1 Introduction

The computation of distance functions between two polygons, in addition to being relevant to computational geometry, has applications in pattern recognition (as described in Cox, Maitre, Minoux and Ribeiro [7]) in the context of determining the optimal matching of two given convex polygons (not necessarily having the same number of vertices, and with a non-empty intersection in the general case) representing, for instance, two distinct series of observations of the same contour or planar object. There it is shown [7] that finding the projection of any vertex of one of the polygons on the other is the basic subproblem to be solved at each step of the algorithm which obtains their optimal matching.

In that reference, the authors use a trivial $O(|P||Q|)$ time algorithm for the computation of a special kind of distance bewteen two convex polygons P and Q (we use $|P|$ to denote the number of vertices of P).

We present in this paper algorithms for the computation of some distance functions between two (possibly intersecting) polygons, both in the convex and nonconvex cases. Among the distances we consider are the Hausdorff distance and that defined by Cox, Maitre, Minoux and Ribeiro [7]. We give an $O(|P| + |Q|)$ time algorithm for the case where P and Q are convex (this is an extension of the previous work of Atallah [1] for the computation of the Hausdorff distance in the case where the two polygons do not intersect). The algorithm is shown to be optimally parallelizable: $O(\log n)$ time with $O(n/\log n)$ processors in the EREW-PRAM model of parallel computation. We also consider the case where P and Q are nonconvex and give an $O(|P||Q| + |P|\log|P| + |Q|\log|Q|)$ time algorithm; such an algorithm does not consist of considering "all possible pairs of vertices/edges" because one of the two points achieving the distance we seek can, in the nonconvex case, be interior to a polygon (and hence there is no obvious enumerative scheme for finding it).

The paper is organized as follows. Section 2 gives the problem statement, along with the basic notation. Section 3 covers the case where P and Q are convex, both in the sequential and parallel settings. Section 4 deals with the case of nonconvex polygons. Section 5 discusses computational results and concludes.

2 Problem Statement

Each of the two input polygons P and Q is given as an ordered sequence of vertices, in (say) clockwise cyclic order (if it has holes then it is specified as many such sequences, one for its outer boundary and one for each hole boundary). For any point p and polygon P, we use $d(p, P)$ to denote the distance from p to P. That is, $d(p, P)$ is zero if p is in the interior of P, otherwise it is the shortest euclidean distance from p to the boundary of P. If p and q are points then $d(p, q)$ is the euclidean distance between them (this is the degenerate case of $P = q$). We use $V(P)$ to denote the set of vertices of polygon P.

With this notation, the distance defined by Cox, Maitre, Minoux and Ribeiro [7] for the case where P and Q are convex, can be expressed as

$$D_{CMMR}(P,Q) = D_{CMMR}(Q,P) = \sum_{p \in V(P)} d^2(p,Q) + \sum_{q \in V(Q)} d^2(q,P),$$

i.e., the sum of the squares of the euclidean distances from each vertex of each polygon to the other polygon. The Hausdorff distance [11] considered by Atallah [1] can be expressed with this notation, as

$$D_H(P,Q) = D_H(Q,P) = \max\{\max_{p \in P} d(p,Q), \max_{q \in Q} d(q,P)\},$$

i.e., the maximum euclidean distance from any point (not necessarily a vertex, and possibly interior) of any polygon to the other polygon. Let p, q be the pair of points $p \in P$ and $q \in Q$ that achieve $D_H(P,Q)$; that is, $d(p,q) = D_H(P,Q)$. When P and Q are convex and none of them contains the other, p and q must belong to the boundaries of P and (respectively) Q, and at least one of them is a vertex of its polygon. It is trivial to see that this is no longer guaranteed when convexity does not hold. For example, when Q is shaped like a thin donut with a large hole in its center, and P is regular (hence convex) and sits in the donut's hole (almost but not quite filling that hole), then $D_H(P,Q) = d(p,Q)$ where p is the center of mass of P (hence is interior to P). See Figure 1.

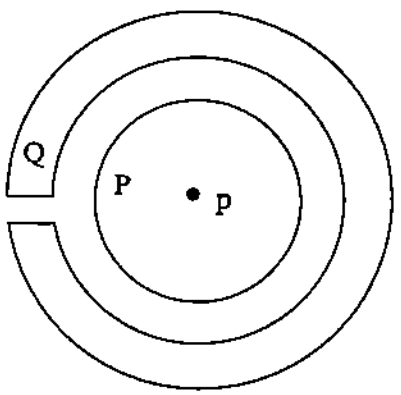

Figure 1. Example where $D_H(P,Q)$ is achieved by an interior point.

Throughout, we use n to denote the total number of vertices of P and Q, i.e., $n = |P| + |Q|$. We use $Interior(P)$ to denote the interior of P, that is, P minus its boundary.

3 Convex Polygons

Throughout this section we assume that P and Q are convex. In this case, for both distance functions of interest, it suffices to compute $d(p,Q)$ and $d(q,P)$ for all $p \in V(P)$, $q \in V(Q)$ (since $D_H(P,Q)$ and $D_{CMMR}(P,Q)$ can both be computed from these with an $O(n)$ extra effort).

We give an algorithm for computing $d(p, Q)$ for all $p \in V(P)$ (using the same algorithm with the roles of P and Q interchanged would give $d(q, P)$ for all $q \in V(Q)$). We assume that P is not entirely contained in Q (the problem is trivial otherwise).

3.1 A Linear Time Sequential Algorithm

We give an $O(n)$ time sequential algorithm as a "warmup" for the parallel algorithm of the next section. First we compute, in linear time, a description of the region of the plane in P or Q but not in the interior of their intersection (call this region C). That is, $C = (P \cup Q) - Interior(P \cap Q)$. The region C consists of a number of "pieces", where each piece is typically not convex, and the boundary of each piece consists of two convex polygonal chains: one coming from the boundary of P (we call it the "P-chain" of that piece), the other coming from the boundary of Q (we call it the "Q-chain" of that piece). A piece has a (clockwise) "predecessor piece" and a "successor piece", in the obvious way. For example, in Figure 2, C consists of 6 pieces, the predecessor of piece A is A', and its successor is piece A''. The computation of C takes linear time as a consequence of the known linear time algorithms for intersecting two convex polygons (e.g., the algorithm in [14]). The vertices of P that are in Q have zero distance to Q and we do not worry about them: we only worry about the other vertices of P (all of which are in C).

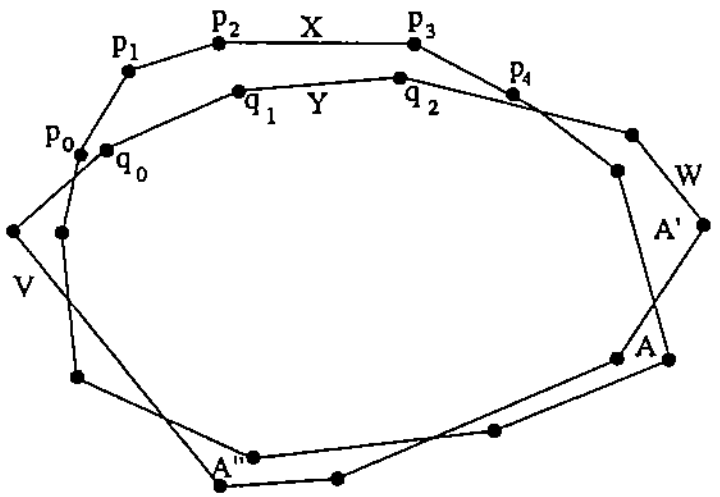

Figure 2. Illustrating the pieces of C.

Let X be the P-chain of a piece $\mathcal{P}$ of C (where X is outside of Q), Y be the Q-chain of $\mathcal{P}$, V be the Q-chain of the predecessor of $\mathcal{P}$ in C, W be the Q-chain of the successor of $\mathcal{P}$ in C. (Note that chains V, Y, W are adjacent and occur in that order along the boundary of Q; see Figure 2.) It easily follows from convexity that, for each vertex p of X, the point q of Q closest to p is on one of $\{V, Y, W\}$. This implies that as far as X is concerned, it suffices to look at Y, V and W only. We only explain how to do it for X and Y in $O(|X| + |Y|)$ time, because for X and V we can do it in $O(|X| + |V|)$ time by using the algorithm of [1] for the

disjoint polygons case, and similarly for X and W in $O(|X| + |W|)$ time (these two cases are essentially like the disjoint-polygons situation—the fact that they "touch" at one point is of no consequence to the algorithm in [1]). For X and Y, we do it in $O(|X| + |Y|)$ time by exploiting the following observation. Let $V(X) = (p_0, \ldots, p_{|X|-1})$, $V(Y) = (q_0, \ldots, q_{|Y|-1})$ (see Figure 2). Let $f(p_i)$ be the point of Y closest to p_i. Observe that the sequence $f(p_0), \ldots, f(p_{|X|-1})$ is sorted along Y. This implies that we can essentially walk along X and Y simultaneously (like the way two sorted lists are merged), computing as we go along for each vertex p_i on X that we encounter the point of Y closest to it. What makes it work is the fact that, as we move monotonically along X, we also move monotonically along Y (during this walk, the distances themselves are, of course, not monotonic: the sequence of distances can switch from increasing to decreasing many times). This walk along X and Y is easily seen to take $O(|X| + |Y|)$ time. This completes the sequential algorithm, which is described in details in Ribeiro and Lifschitz [17] along with computational results. The parallel algorithm is discussed next.

3.2 Optimal EREW-PRAM Algorithm

Recall that the PRAM is the shared-memory model of parallel computation in which the processors, which operate synchronously, share a common memory that they access concurrently. The EREW version of the PRAM is the one where at any time step, no two processors can attempt to simultaneously access the same memory cell (and hence is the weakest version of the PRAM, since other versions allow concurrent reading and/or writing). We sketch how the algorithm of the previous subsection can be simulated on the EREW-PRAM model so that it runs in $O(\log n)$ time with $O(n/\log n)$ processors. Before giving the details, we observe that these bounds are worst-case optimal for this model. The *time* × *processors* product is optimal because (i) these same bounds are known to be optimal for the problem of computing the logical OR of n given boolean variables [6], and (ii) the latter problem is reducible to our problem without any cost (that is, in zero time and with zero processors). The no-cost "reduction" is as follows: we imagine Q to be a regular n-gon (hence convex) centered at the origin of coordinates, and we encode the n boolean variables in P by first considering P to be a slightly "shrunk" version of Q (also centered at the origin), and then "perturbing" each vertex of P so that it encodes one of the n boolean variables (a "1" is encoded by moving the vertex slightly outside of Q, a "0" by keeping it inside of Q). Note that P and Q are not constructed explicitly: the "shrinkage factor" (for getting P from Q) and the n boolean variables are a complete implicit description of P and Q (i.e., from them one processor can get any vertex of P or Q in constant time).

We now turn our attention to the parallel algorithm. It will be convenient to analyze it using the time and work (i.e., number of operations) complexities. The processor complexity is deduced from these using Brent's theorem [3], which states that any synchronous parallel algorithm taking time $\mathcal{T}$ that consists of a total of $\mathcal{W}$ operations can be simulated by $\mathcal{P}$ processors in time $O((\mathcal{W}/\mathcal{P}) + \mathcal{T})$. There are actually two qualifications to Brent's theorem before one can apply it to a PRAM: (i) at the beginning of the i-th parallel

step, we must be able to compute the amount of work W_i done by that step, in time $O(W_i/\mathcal{P})$ and with $\mathcal{P}$ processors, and (ii) we must know how to assign each processor to its task. Both qualifications (i) and (ii) to Brent's theorem will be easily satisfied in our case. Consequently, we can henceforth concern ourselves with achieving $\mathcal{W}= O(n)$ work in time $\mathcal{T}= O(\log n)$, using $\mathcal{P}= n$ processors. Using n processors rather than $n/\log n$ will simplify our exposition, and Brent's theorem guarantees that we can decrease the number of processors to $O(n/\log n)$ without any deterioration in the time complexity.

The EREW-PRAM parallel algorithm for obtaining the Hausdorff distance through the computation of the point of P furthest from Q is structured as follows (the algorithm for the computation of D_{CMMR} would be quite similar and is omitted here):

1. If P is inside Q then all points of P are at a distance of zero from Q, and we can stop. Otherwise we proceed to the next step.
2. Compute $P \cap Q$, the region C, and break C up into its constituent pieces.
3. Process the pieces of C in parallel. That is, if for a certain piece $\mathcal{P}$ we let X, V, Y, W be as in the previous subsection, we need to compute for each vertex of X the point of Y (respectively V, W) closest to it. We shall explain, after this outline of the algorithm, how this is done for Y, and we later point out how the computation for V and W can be reduced to a constant number of computations each of which is similar to the computation for Y.
4. Collect answers by finding the vertex of $C \cap P$ that is farthest from Q.

There are two main bottlenecks to implementing the above steps within the desired parallel bounds: (i) computing the description of $P \cap Q$ (step 2), and (ii) the "walks" along X and Y, X and V, and X and W (step 3). (The other steps are trivial to perform in logarithmic time and linear work.) Computing $P \cap Q$ (hence C and its pieces) turns out to be easy, while parallelizing the "walk" along X and Y is more interesting. We use the fact that it is known how to merge two sorted sequences within these complexity bounds, and also a recent result of Chen [5] for computing the convex hull of a sorted point set in $O(\log n)$ time and $O(n/\log n)$ processors on the EREW-PRAM.

Computing $P \cap Q$ is the dual [16] of the problem of computing the convex hull of the union of two sorted point sets, and hence it can be done within the desired complexity bounds (by first merging the two sets and then using Chen's above-mentioned convex hull algorithm). Therefore, it suffices to show that each of the three walks involving X (i.e., X with each of Y, V, and W) can be parallelized so that it takes logarithmic time and linear work. We begin with the easier of the three cases, that for the pair X, Y, in which the goal is to achieve $O(\log(|X| + |Y|))$ time and $O(|X| + |Y|)$ work.

Parallelizing the walk along the vertices of X and Y requires a careful look at the way the above-mentioned sequential walk along X and Y works. Suppose that, in the sequential algorithm, we are at $x \in X$ and $y \in Y$. We decide whether to take a step along X or Y as follows:

1. Let s be the segment joining y to its predecessor vertex in Y, let L be the line containing s, and let z be the foot of the perpendicular to L from x. If $z \in s - \{y\}$ then we set $f(x) = z$, we move along X to the successor of x, and we remain at y in Y. If z is not in $s - \{y\}$ then we proceed to (2) below.

2. Let T be the perpendicular at y to the segment xy. If T is tangent to Q at y then we set $f(x) = y$ and we move along X to the successor of x. Otherwise we proceed to (3) below.

3. If T is not tangent to Q at y then we move along Y to the successor of y, and remain at x in X. See Figure 3a, and note that in this case the successor of y in Y (call it y') is on the same side of T as x (this follows from the way the sequential algorithm works, as assuming y' and x to be on opposite sides of T quickly leads to a contradiction with the way the algorithm works).

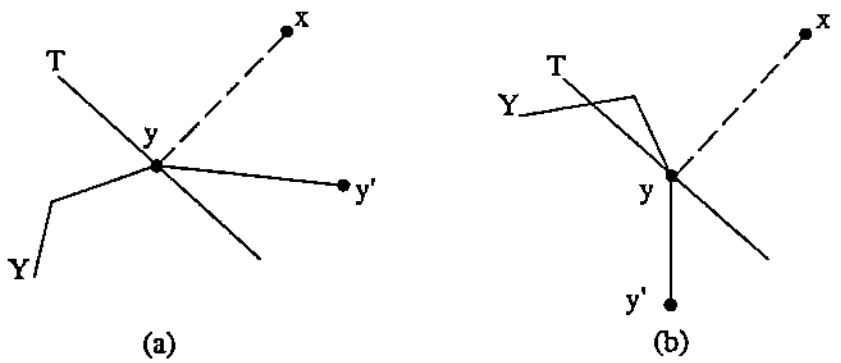

Figure 3. Illustrating the relative position of T and y'.

It naturally suggests itself that one should mimic the above sequential computation by using an optimal parallel merging algorithm (such as that of [2] or of [12]), and using T to "compare" x and y. However, there are complications: in case T is not tangent, we might well have y' and x on opposite sides of T (see Figure 3b). When the parallel merging algorithm compares an $x \in X$ to a $y \in Y$, it expects to know the outcome of such a comparison. We would like the parallel merging of X and Y to produce a "merged" sequence $X \cup Y$ in which each $x \in X$ has $f(x)$ either already computed by the merging procedure, or determined by the two elements of Y that are before and after it in $X \cup Y$ (that is, if these elements are y' and y'', then $f(x)$ is either y', or y'', or the foot of the perpendicular from x to the segment $y'y''$). This property (assuming we can somehow achieve it) would be sufficient, because the merging procedures of [2] and [12] produce, in addition to the merged sequence, the rank of each element of X in Y (and vice-versa), so that the above-mentioned y' and y'' are available in constant time for each $x \in X$, and hence it is trivial to obtain all the $f(x)$'s with a postprocessing step that takes constant time with $\alpha = |X| + |Y|$ processors or, alternatively, $O(\log \alpha)$ time and $O(\alpha)$ work. Hence from now on we focus on the problem of computing an $X \cup Y$ that has the above-mentioned desired property. One

can prove that this property of $X \cup Y$ will hold so long as a comparison "$x : y$" is treated as follows by the parallel merging algorithm (as before, T is the perpendicular to segment xy at y):

1. Let s be the segment joining y to its predecessor in Y, let L be the line containing s, and let z be the foot of the perpendicular to L from x. If $z \in s - \{y\}$ then the merging algorithm behaves as if the outcome of the comparison had "$x < y$" as its outcome, otherwise it behaves as in (2)–(4) below.

2. If T is tangent to Q at y then the parallel merging algorithm sets $f(x) = y$ and behaves as if the outcome of the "comparison" of x with y had "$x = y$" as its outcome.

3. If T is not tangent to Q at y and if the successor of y in Y is on the same side of T as x, then the parallel merging algorithm behaves as if the outcome of the "comparison" of x with y had "$x > y$" as its outcome.

4. If T is not tangent to Q at y and if the successor of y in Y is on the opposite side of T relative to x, then the parallel merging algorithm behaves as if the outcome of the "comparison" of x with y had "$x < y$" as its outcome.

Note that the above scheme does *not* guarantee that every $f(x)$ will have been computed for every $x \in X$ when the parallel merge terminates (that is, the postprocessing stage we mentioned earlier is in fact needed). The fact that the $X \cup Y$ produced by this scheme has the desired property is a consequence of the fact that our rules for the outcomes of "comparisons" of x and y are as if we were comparing y to $f(x)$ rather than to x. That is, it is as if we were in effect merging Y with the (still unknown) sequence $f(X)$ rather than with X itself.

The computation for the pair X, V (i.e., when V is playing the role of Y) must be handled somewhat differently, since in the sequential algorithm the direction of the walk along V can change, and hence the analogy with parallel merging seems to break down. However, we can reduce the computation for the pair X, V to four computations, each of which is similar to the computation for the pair X, Y. We do this as follows. We compute the common tangent between X and V (there are many such tangents; we are referring to the one that does not intersect $P \cap Q$). This tangent can be computed in $O(\log(|X| + |V|))$ time by a single processor [15]. The point at which this common tangent touches X (resp., V) partitions X (resp., V) into two chains X_1, X_2 (resp., V_1, V_2). Clearly, to solve the problem defined by X, V it suffices to solve the four subproblems defined by the four pairs X_i, V_j for $i, j \in \{1, 2\}$. (In fact it is enough to solve three of these subproblems – we do not elaborate on this, however, in order not to clutter the exposition.) But each such pair X_i, V_j is a problem similar to that for the X, Y pair, which we have already explained how to solve. Hence the X, V case can be handled within complexity bounds similar to those for the X, Y case, namely, $O(\log(|X| + |V|))$ time and $O(|X| + |V|)$ work.

Of course, the problem for the pair X, W is handled similarly to that for X, V.

4 Arbitrary P and Q

In this section we consider the Hausdorff distance computation in the general case where P and Q are not necessarily convex, can have holes, and possibly intersect. Essentially the same technique solves the problem for other distance measures, including any of the natural generalizations of D_{CMMR} to nonconvex polygons (for example, replacing in the definition of D_{CMMR} each $d(p,Q)$ by $d(p,Boundary(Q))$). Since the procedure for such a D_{CMMR} would be quite similar to the one we give below for D_H, we choose to omit its detailed specification and focus instead on D_H, the Hausdorff case.

We only compute the point of P farthest from Q (using the same algorithm with the roles of P and Q interchanged gives the point of Q farthest from P). It is trivial to see that this point can be interior to P (as in Figure 1). However, although there are apparently infinitely many points of P we need to consider, an $O(|P||Q|+|P|\log|P|+|Q|\log|Q|)$ time algorithm is obtained as follows.

Triangulate P, and compute the Voronoi Diagram [16] of the segments of Q (call it $Vor(Q)$). These can be done in time $O(|P|\log|P|)$ and (respectively) $O(|Q|\log|Q|)$ by well-known algorithms (e.g., [10] for triangulation and [18] for Voronoi). We then modify $Vor(Q)$ so that it becomes a planar subdivision $S(Q)$ obtained by (i) coalescing, in the $Vor(Q)$ planar subdivision, all the faces that are inside Q into a single (nonconvex, possibly with holes) face, and then (ii) triangulating Q. That is, the portion of $S(Q)$ outside of Q is identical to $Vor(Q)$, and the portion of $S(Q)$ inside of Q is the triangulation of Q. Of course we have $|S(Q)| = O(|Q|)$.

Let T be any triangle in the triangulation of P. It clearly suffices to give an $O(|Q|)$ time algorithm for getting the point of T that is farthest from Q. This is done as follows.

1. Intersect T with the planar subdivision defined by $S(Q)$, inducing a subdivision of T into no more than $\ell = O(|Q|)$ pieces $T_1,\ldots,T_\ell$ (possibly $\ell = 1$, in which case $T_1 = T$).
   Implementation Note. Computing the T_i's is done in $O(|Q|)$ time by locating a vertex of T in the $S(Q)$ planar subdivision and then "cutting" out of that planar subdivision the triangular region T by tracing the boundary of T and computing this boundary's intersection(s) with the edges of $S(Q)$. The details are straightforward and are omitted.
   Comment. Observe that although the size of a particular T_i need not be $O(1)$, we have $\sum_{1\le i\le \ell}|T_i| = O(|Q|)$. Therefore to achieve the $O(|Q|)$ bound we claim for processing T it suffices to show that, for each T_i, the point of T_i that is farthest from Q can be found in $O(|T_i|)$ time. This is done in the next two steps.

2. For each T_i that is inside Q, any point of T_i has zero distance to Q.
   Comment. Whether T_i is inside Q is obtained in Step 1, as a byproduct of the computation of T_i.

3. For each T_i that is not inside Q, computing the point of T_i furthest from Q can be done by computing the largest distance between any vertex of T_i and the segment or

vertex of Q that corresponds to the face of $Vor(Q)$ containing T_i. Thus one needs only to do one constant-time computation for each vertex of T_i.

The above clearly implies an $O(|P||Q| + |P|\log|P| + |Q|\log|Q|)$ time algorithm. Note that a consequence of Step 3 is that, if P is not contained in Q, the point of P that is farthest from Q is a vertex of some T_i (in fact a vertex of $Vor(Q)$).

5 Summary and Conclusions

We gave a linear time sequential algorithm and an $O(\log n)$ time and $O(n/\log n)$ processor EREW-PRAM parallel algorithm for the computation of some distance functions between two (possibly intersecting) convex polygons. We also gave an $O(n^2)$ sequential algorithm for the case when the polygons are not convex. The distance functions considered are motivated by applications in robot vision, pattern recognition, and contour fitting.

Algorithmic details and computational results concerning the sequential algorithm described in Section 3.1 are presented in [17]. It was noticed during the computational experiences reported in Cox, Maitre, Minoux and Ribeiro [7] that most of the computational times observed for the solution of the problem of optimal matching of convex polygons corresponds to computations of the distance function between them. The linear time sequential algorithm described in Section 3.1 can then be used to considerably speed up the approach proposed in that paper. Furthermore, the algorithm presented in Section 4 for arbitrary polygons can be used for the optimal matching of non convex polygons corresponding to more detailed contour representations, in the context of the application described in [7] (i.e., instead of considering just the convex hulls of the contours to be matched).

A very useful result would be the extension to convex polyhedra. Such a 3-dimensioanl solution is virtually certain to involve the result of Chazelle [4] on linear-time computation of the intersection of convex polyhedra, and the Dobkin-Kirkpatrick technique of hierarchically representing polyhedra [8, 9]. Another interesting line of research, pursued in [13] for point sets, is to find a translation (or other motion) that, when applied to one set, minimizes the distance function.

References

[1] M. J. Atallah, "A Linear Time Algorithm for the Hausdorff Distance between Convex Polygons", *Information Processing Letters* 17 (1983), 207-209.

[2] G. Bilardi and A. Nicolau, "Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines", *SIAM Journal on Computing* 18 (1989), 216-228.

[3] R. P. Brent. "The parallel evaluation of general arithmetic expressions", *Journal of the ACM* 21 (2) (1974), 201-206.

[4] B. M. Chazelle, "An Optimal Algorithm for Intersecting Three-Dimensional Convex Polyhedra", *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, 1989, Research Triangle, NC, 586-591.

[5] D. Z. Chen. "An EREW-PRAM algorithm for the convex hull of a sorted point set", TR 89-928, Department of Computer Science, Purdue University, November 1989.

[6] S. Cook and C. Dwork, "Bounds on the Time for Parallel RAMs to Compute Simple Functions", *Proceedings of the 14th ACM Symposium on Theory of Computation*, 1982, San Francisco, CA, 231-233.

[7] P. Cox, H. Maitre, M. Minoux and C. C. Ribeiro, "Optimal Matching of Convex Polygons", *Pattern Recognition Letters* 9 (1989), 327-334.

[8] D. P. Dobkin and D. G. Kirkpatrick, "Fast Detection of Polyhedral Intersection", *Theoretical Computer Science* 27 (1983), 241-253.

[9] D. P. Dobkin and D. G. Kirkpatrick, "A Linear Algorithm for Determining the Separation of Convex Polyhedra", *Journal of Algorithms* 6 (1985), 381-392.

[10] M. R. Garey, D. S. Johnson, F. P. Preparata and R. E. Tarjan, "Triangulating a Simple Polygon", *Information Processing Letters* 7 (1978), 175-179.

[11] B. Grunbaum, *Convex Polytopes*, Wiley, New York, 1967.

[12] T. Hagerup and C. Rub. "Optimal merging and sorting on the EREW PRAM", *Information Processing Letters* 33 (1989), 181-185.

[13] D. P. Huttenlocher and K. Kedem, "Computing the Minimum Hausdorff Distance for Point Sets Under Translation", *Proceedings of the 6th Annual ACM Symposium on Computational Geometry*, 1990, Berkeley, CA, 340–349.

[14] J. O'Rourke, C.-B. Chien, T. Olson and D. Naddor, "A New Linear Time Algorithm for Intersecting Convex Polygons", *Computer Graphics and Image Processing* 19 (1982), 384-391.

[15] M. H. Overmars and J. Van Leeuwen, "Maintenance of Configurations in the Plane", *Journal of Computer and Systems Sciences* 23 (1981), 166-204.

[16] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[17] C. C. Ribeiro and S. Lifschitz, "A Linear Time Algorithm for the Computation of Some Distance Functions between Convex Polygons", Les Cahiers du GERAD G-90-23, École des Hautes Études Commerciales, Montréal, Canada, February 1990.

[18] C. K. Yap, "An $O(n \log n)$ Algorithm for the Voronoi Diagram of a Set of Simple Curve Segments", *Discrete and Computational Geometry* 2 (1987), 365-393.